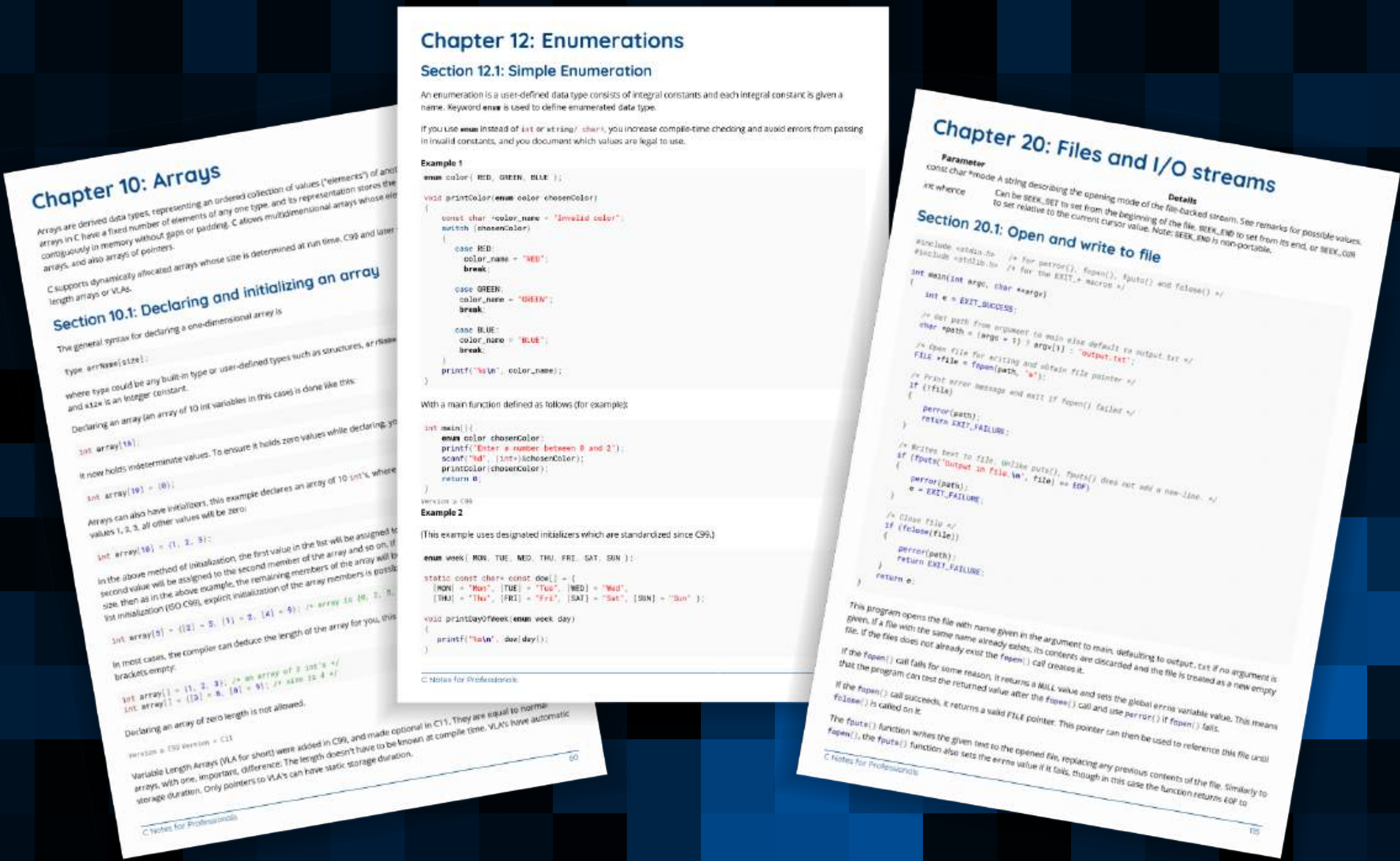
C

Notes for Professionals



300+ pages

of professional hints and tricks

GoalKicker.com

Free Programming Books

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/CBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with C Language

Version	Standard	Publication Date
K&R	n/a	1978-02-22
C89	ANSI X3.159-1989	1989-12-14
C90	ISO/IEC 9899:1990	1990-12-20
C95	ISO/IEC 9899/AMD1:1995	1995-03-30
C99	ISO/IEC 9899:1999	1999-12-16
C11	ISO/IEC 9899:2011	2011-12-15

Section 1.1: Hello World

To create a simple C program which prints *"Hello, World"* on the screen, use a text editor to create a new file (e.g. `hello.c` — the file extension must be `.c`) containing the following source code:

hello.c

```
#include <stdio.h>

int main(void)
{
    puts("Hello, World");
    return 0;
}
```

Live demo on Coliru

Let's look at this simple program line by line

```
#include <stdio.h>
```

This line tells the compiler to include the contents of the standard library header file `stdio.h` in the program. Headers are usually files containing function declarations, macros and data types, and you must include the header file before you use them. This line includes `stdio.h` so it can call the function `puts()`.

See more about headers.

```
int main(void)
```

This line starts the definition of a function. It states the name of the function (`main`), the type and number of arguments it expects (`void`, meaning none), and the type of value that this function returns (`int`). Program execution starts in the `main()` function.

```
{
    …
}
```

The curly braces are used in pairs to indicate where a block of code begins and ends. They can be used in a lot of ways, but in this case they indicate where the function begins and ends.

```
    puts("Hello, World");
```

This line calls the `puts()` function to output text to standard output (the screen, by default), followed by a newline.

The string to be output is included within the parentheses.

`"Hello, World"` is the string that will be written to the screen. In C, every string literal value must be inside the double quotes `"…"`.

See more about strings.

In C programs, every statement needs to be terminated by a semi-colon (i.e. `;`).

```
    return 0;
```

When we defined `main()`, we declared it as a function returning an `int`, meaning it needs to return an integer. In this example, we are returning the integer value 0, which is used to indicate that the program exited successfully. After the `return 0;` statement, the execution process will terminate.

Editing the program

Simple text editors include `vim` or `gedit` on Linux, or `Notepad` on Windows. Cross-platform editors also include `Visual Studio Code` or `Sublime Text`.

The editor must create plain text files, not RTF or other any other format.

Compiling and running the program

To run the program, this source file (`hello.c`) first needs to be compiled into an executable file (e.g. `hello` on Unix/Linux system or `hello.exe` on Windows). This is done using a compiler for the C language.

See more about compiling

Compile using GCC

GCC (GNU Compiler Collection) is a widely used C compiler. To use it, open a terminal, use the command line to navigate to the source file's location and then run:

```
gcc hello.c -o hello
```

If no errors are found in the the source code (`hello.c`), the compiler will create a **binary file**, the name of which is given by the argument to the `-o` command line option (`hello`). This is the final executable file.

We can also use the warning options `-Wall -Wextra -Werror`, that help to identify problems that can cause the program to fail or produce unexpected results. They are not necessary for this simple program but this is way of adding them:

```
gcc -Wall -Wextra -Werror -o hello hello.c
```

Using the clang compiler

To compile the program using `clang` you can use:

```
clang -Wall -Wextra -Werror -o hello hello.c
```

By design, the `clang` command line options are similar to those of GCC.

Using the Microsoft C compiler from the command line

If using the Microsoft `cl.exe` compiler on a Windows system which supports [Visual Studio] and if all environment variables are set, this C example may be compiled using the following command which will produce an executable `hello.exe` within the directory the command is executed in (There are warning options such as `/W3` for `cl`, roughly analogous to `-Wall` etc for GCC or clang).

```
cl hello.c
```

Executing the program

Once compiled, the binary file may then be executed by typing `./hello` in the terminal. Upon execution, the compiled program will print `Hello, World`, followed by a newline, to the command prompt.

Section 1.2: Original "Hello, World!" in K&R C

The following is the original "Hello, World!" program from the book The C Programming Language by Brian Kernighan and Dennis Ritchie (Ritchie was the original developer of the C programming language at Bell Labs), referred to as "K&R":

Version = K&R

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Notice that the C programming language was not standardized at the time of writing the first edition of this book (1978), and that this program will probably not compile on most modern compilers unless they are instructed to accept C90 code.

This very first example in the K&R book is now considered poor quality, in part because it lacks an explicit return type for `main()` and in part because it lacks a `return` statement. The 2nd edition of the book was written for the old C89 standard. In C89, the type of `main` would default to `int`, but the K&R example does not return a defined value to the environment. In C99 and later standards, the return type is required, but it is safe to leave out the `return` statement of `main` (and only `main`), because of a special case introduced with C99 5.1.2.2.3 — it is equivalent to returning 0, which indicates success.

The recommended and most portable form of `main` for hosted systems is `int main (void)` when the program does not use any command line arguments, or `int main(int argc, char **argv)` when the program does use the command line arguments.

C90 §5.1.2.2.3 **Program termination**

> A return from the initial call to the `main` function is equivalent to calling the `exit` function with the value returned by the `main` function as its argument. If the `main` function executes a return that specifies no value, the termination status returned to the host environment is undefined.

C90 §6.6.6.4 **The `return` statement**

> If a `return` statement without an expression is executed, and the value of the function call is used by the

> caller, the behavior is undefined. Reaching the `}` that terminates a function is equivalent to executing a `return` statement without an expression.

C99 §5.1.2.2.3 **Program termination**

> If the return type of the `main` function is a type compatible with `int`, a return from the initial call to the `main` function is equivalent to calling the `exit` function with the value returned by the `main` function as its argument; reaching the `}` that terminates the `main` function returns a value of 0. If the return type is not compatible with `int`, the termination status returned to the host environment is unspecified.

Chapter 2: Comments

Comments are used to indicate something to the person reading the code. Comments are treated like a blank by the compiler and do not change anything in the code's actual meaning. There are two syntaxes used for comments in C, the original `/* */` and the slightly newer `//`. Some documentation systems use specially formatted comments to help produce the documentation for code.

Section 2.1: Commenting using the preprocessor

Large chunks of code can also be "commented out" using the preprocessor directives `#if 0` and `#endif`. This is useful when the code contains multi-line comments that otherwise would not nest.

```
#if 0 /* Starts the "comment", anything from here on is removed by preprocessor */

/* A large amount of code with multi-line comments */
int foo()
{
    /* lots of code */
    ...

    /* ... some comment describing the if statement ... */
    if (someTest) {
        /* some more comments */
        return 1;
    }

    return 0;
}

#endif /* 0 */

/* code from here on is "uncommented" (included in compiled executable) */
...
```

Section 2.2: /* */ delimited comments

A comment starts with a forward slash followed immediately by an asterisk (`/*`), and ends as soon as an asterisk immediately followed by a forward slash (`*/`) is encountered. Everything in between these character combinations is a comment and is treated as a blank (basically ignored) by the compiler.

```
/* this is a comment */
```

The comment above is a single line comment. Comments of this `/*` type can span multiple lines, like so:

```
/* this is a
multi-line
comment */
```

Though it is not strictly necessary, a common style convention with multi-line comments is to put leading spaces and asterisks on the lines subsequent to the first, and the `/*` and `*/` on new lines, such that they all line up:

```
/*
 * this is a
 * multi-line
 * comment
```

```
 */
```

The extra asterisks do not have any functional effect on the comment as none of them have a related forward slash.

These `/*` type of comments can be used on their own line, at the end of a code line, or even within lines of code:

```
/* this comment is on its own line */
if (x && y) { /*this comment is at the end of a line */
    if ((complexCondition1) /* this comment is within a line of code */
            && (complexCondition2)) {
        /* this comment is within an if, on its own line */
    }
}
```

Comments cannot be nested. This is because any subsequent `/*` will be ignored (as part of the comment) and the first `*/` reached will be treated as ending the comment. The comment in the following example *will not work*:

```
/* outer comment, means this is ignored => /* attempted inner comment */ <= ends the comment, not
this one => */
```

To comment blocks of code that contain comments of this type, that would otherwise be nested, see the Commenting using the preprocessor example below

Section 2.3: // delimited comments

Version ≥ C99

C99 introduced the use of C++-style single-line comments. This type of comment starts with two forward slashes and runs to the end of a line:

```
// this is a comment
```

This type of comment does not allow multi-line comments, though it is possible to make a comment block by adding several single line comments one after the other:

```
// each of these lines are a single-line comment
// note how each must start with
// the double forward-slash
```

This type of comment may be used on its own line or at the end of a code line. However, because they run *to the end of the line*, they may *not* be used within a code line

```
// this comment is on its own line
if (x && y) { // this comment is at the end of a line
    // this comment is within an if, on its own line
}
```

Section 2.4: Possible pitfall due to trigraphs

Version ≥ C99

While writing `//` delimited comments, it is possible to make a typographical error that affects their expected operation. If one types:

```c
int x = 20;  // Why did I do this??/
```

The `/` at the end was a typo but now will get interpreted into `\`. This is because the `??/` forms a trigraph.

The `??/` trigraph is actually a longhand notation for `\`, which is the line continuation symbol. This means that the compiler thinks the next line is a continuation of the current line, that is, a continuation of the comment, which may not be what is intended.

```c
int foo = 20; // Start at 20 ??/
int bar = 0;

// The following will cause a compilation error (undeclared variable 'bar')
// because 'int bar = 0;' is part of the comment on the preceding line
bar += foo;
```

Chapter 3: Data Types

Section 3.1: Interpreting Declarations

A distinctive syntactic peculiarity of C is that declarations mirror the use of the declared object as it would be in a normal expression.

The following set of operators with identical precedence and associativity are reused in declarators, namely:

- the unary `*` "dereference" operator which denotes a pointer;
- the binary `[]` "array subscription" operator which denotes an array;
- the (1+n)-ary `()` "function call" operator which denotes a function;
- the `()` grouping parentheses which override the precedence and associativity of the rest of the listed operators.

The above three operators have the following precedence and associativity:

Operator	Relative Precedence	Associativity
`[]` (array subscription)	1	Left-to-right
`()` (function call)	1	Left-to-right
`*` (dereference)	2	Right-to-left

When interpreting declarations, one has to start from the identifier outwards and apply the adjacent operators in the correct order as per the above table. Each application of an operator can be substituted with the following English words:

Expression	Interpretation
`thing[X]`	an array of size `X` of...
`thing(t1, t2, t3)`	a function taking `t1`, `t2`, `t3` and returning...
`*thing`	a pointer to...

It follows that the beginning of the English interpretation will always start with the identifier and will end with the type that stands on the left-hand side of the declaration.

Examples

```
char *names[20];
```

`[]` takes precedence over `*`, so the interpretation is: `names` is an array of size 20 of a pointer to `char`.

```
char (*place)[10];
```

In case of using parentheses to override the precedence, the `*` is applied first: `place` is a pointer to an array of size 10 of `char`.

```
int fn(long, short);
```

There is no precedence to worry about here: `fn` is a function taking `long`, `short` and returning `int`.

```
int *fn(void);
```

The `()` is applied first: `fn` is a function taking `void` and returning a pointer to `int`.

```
int (*fp)(void);
```

Overriding the precedence of `()`: `fp` is a pointer to a function taking `void` and returning `int`.

```
int arr[5][8];
```

Multidimensional arrays are not an exception to the rule; the `[]` operators are applied in left-to-right order according to the associativity in the table: `arr` is an array of size 5 of an array of size 8 of `int`.

```
int **ptr;
```

The two dereference operators have equal precedence, so the associativity takes effect. The operators are applied in right-to-left order: `ptr` is a pointer to a pointer to an `int`.

Multiple Declarations

The comma can be used as a separator (*not* acting like the comma operator) in order to delimit multiple declarations within a single statement. The following statement contains five declarations:

```
int fn(void), *ptr, (*fp)(int), arr[10][20], num;
```

The declared objects in the above example are:

- `fn`: a function taking `void` and returning `int`;
- `ptr`: a pointer to an `int`;
- `fp`: a pointer to a function taking `int` and returning `int`;
- `arr`: an array of size 10 of an array of size 20 of `int`;
- `num`: `int`.

Alternative Interpretation

Because declarations mirror use, a declaration can also be interpreted in terms of the operators that could be applied over the object and the final resulting type of that expression. The type that stands on the left-hand side is the final result that is yielded after applying all operators.

```
/*
 * Subscripting "arr" and dereferencing it yields a "char" result.
 * Particularly: *arr[5] is of type "char".
 */
char *arr[20];

/*
 * Calling "fn" yields an "int" result.
 * Particularly: fn('b') is of type "int".
 */
int fn(char);

/*
 * Dereferencing "fp" and then calling it yields an "int" result.
 * Particularly: (*fp)() is of type "int".
 */
int (*fp)(void);

/*
 * Subscripting "strings" twice and dereferencing it yields a "char" result.
 * Particularly: *strings[5][15] is of type "char"
 */
char *strings[10][20];
```

Section 3.2: Fixed Width Integer Types (since C99)

Version ≥ C99

The header **<stdint.h>** provides several fixed-width integer type definitions. These types are *optional* and only provided if the platform has an integer type of the corresponding width, and if the corresponding signed type has a two's complement representation of negative values.

See the remarks section for usage hints of fixed width types.

```
/* commonly used types include */
uint32_t u32 = 32; /* exactly 32-bits wide */

uint8_t u8 = 255;  /* exactly 8-bits wide */

int64_t i64 = -65  /* exactly 64 bit in two's complement representation */
```

Section 3.3: Integer types and constants

Signed integers can be of these types (the `int` after `short`, or `long` is optional):

```
signed char c = 127; /* required to be 1 byte, see remarks for further information. */
signed short int si = 32767; /* required to be at least 16 bits. */
signed int i = 32767; /* required to be at least 16 bits */
signed long int li = 2147483647; /* required to be at least 32 bits. */
```

Version ≥ C99

```
signed long long int li = 2147483647; /* required to be at least 64 bits */
```

Each of these signed integer types has an unsigned version.

```
unsigned int i = 65535;
unsigned short = 2767;
unsigned char = 255;
```

For all types but `char` the `signed` version is assumed if the `signed` or `unsigned` part is omitted. The type `char` constitutes a third character type, different from `signed char` and `unsigned char` and the signedness (or not) depends on the platform.

Different types of integer constants (called *literals* in C jargon) can be written in different bases, and different width, based on their prefix or suffix.

```
/* the following variables are initialized to the same value: */
int d = 42;   /* decimal constant (base10) */
int o = 052;  /* octal constant (base8) */
int x = 0xaf; /* hexadecimal constants (base16) */
int X = 0XAf; /* (letters 'a' through 'f' (case insensitive) represent 10 through 15) */
```

Decimal constants are always `signed`. Hexadecimal constants start with `0x` or `0X` and octal constants start just with a `0`. The latter two are `signed` or `unsigned` depending on whether the value fits into the signed type or not.

```
/* suffixes to describe width and signedness : */
long int i = 0x32; /* no suffix represent int, or long int */
unsigned int ui = 65535u; /* u or U represent unsigned int, or long int */
long int li = 65536l; /* l or L represent long int */
```

Without a suffix the constant has the first type that fits its value, that is a decimal constant that is larger than

`INT_MAX` is of type `long` if possible, or `long long` otherwise.

The header file **`<limits.h>`** describes the limits of integers as follows. Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown below, with the same sign.

Macro	Type	Value
`CHAR_BIT`	smallest object that is not a bit-field (byte)	8
`SCHAR_MIN`	`signed char`	-127 / -(27 - 1)
`SCHAR_MAX`	`signed char`	+127 / 27 - 1
`UCHAR_MAX`	`unsigned char`	255 / 28 - 1
`CHAR_MIN`	`char`	see below
`CHAR_MAX`	`char`	see below
`SHRT_MIN`	`short int`	-32767 / -(215 - 1)
`SHRT_MAX`	`short int`	+32767 / 215 - 1
`USHRT_MAX`	`unsigned short int`	65535 / 216 - 1
`INT_MIN`	`int`	-32767 / -(215 - 1)
`INT_MAX`	`int`	+32767 / 215 - 1
`UINT_MAX`	`unsigned int`	65535 / 216 - 1
`LONG_MIN`	`long int`	-2147483647 / -(231 - 1)
`LONG_MAX`	`long int`	+2147483647 / 231 - 1
`ULONG_MAX`	`unsigned long int`	4294967295 / 232 - 1

Version ≥ C99

Macro	Type	Value
`LLONG_MIN`	`long long int`	-9223372036854775807 / -(263 - 1)
`LLONG_MAX`	`long long int`	+9223372036854775807 / 263 - 1
`ULLONG_MAX`	`unsigned long long int`	18446744073709551615 / 264 - 1

If the value of an object of type `char` sign-extends when used in an expression, the value of `CHAR_MIN` shall be the same as that of `SCHAR_MIN` and the value of `CHAR_MAX` shall be the same as that of `SCHAR_MAX` . If the value of an object of type `char` does not sign-extend when used in an expression, the value of `CHAR_MIN` shall be 0 and the value of `CHAR_MAX` shall be the same as that of `UCHAR_MAX`.

Version ≥ C99

The C99 standard added a new header, **`<stdint.h>`**, which contains definitions for fixed width integers. See the fixed width integer example for a more in-depth explanation.

Section 3.4: Floating Point Constants

The C language has three mandatory real floating point types, `float`, `double`, and `long double`.

```
float f = 0.314f;        /* suffix f or F denotes type float */
double d = 0.314;        /* no suffix denotes double */
long double ld = 0.314l; /* suffix l or L denotes long double */

/* the different parts of a floating point definition are optional */
double x = 1.; /* valid, fractional part is optional */
double y = .1; /* valid, whole-number part is optional */

/* they can also defined in scientific notation */
double sd = 1.2e3; /* decimal fraction 1.2 is scaled by 10^3, that is 1200.0 */
```

The header `<float.h>` defines various limits for floating point operations.

Floating point arithmetic is implementation defined. However, most modern platforms (arm, x86, x86_64, MIPS) use IEEE 754 floating point operations.

C also has three optional complex floating point types that are derived from the above.

Section 3.5: String Literals

A string literal in C is a sequence of chars, terminated by a literal zero.

```
char* str = "hello, world"; /* string literal */

/* string literals can be used to initialize arrays */
char a1[] = "abc"; /* a1 is char[4] holding {'a','b','c','\0'} */
char a2[4] = "abc"; /* same as a1 */
char a3[3] = "abc"; /* a1 is char[3] holding {'a','b','c'}, missing the '\0' */
```

String literals are **not modifiable** (and in fact may be placed in read-only memory such as .rodata). Attempting to alter their values results in undefined behaviour.

```
char* s = "foobar";
s[0] = 'F'; /* undefined behaviour */

/* it's good practice to denote string literals as such, by using `const` */
char const* s1 = "foobar";
s1[0] = 'F'; /* compiler error! */
```

Multiple string literals are concatenated at compile time, which means you can write construct like these.

Version < C99

```
/* only two narrow or two wide string literals may be concatenated */
char* s = "Hello, " "World";
```

Version ≥ C99

```
/* since C99, more than two can be concatenated */
/* concatenation is implementation defined */
char* s1 = "Hello" ", " "World";

/* common usages are concatenations of format strings */
char* fmt = "%" PRId16; /* PRId16 macro since C99 */
```

String literals, same as character constants, support different character sets.

```
/* normal string literal, of type char[] */
char* s1 = "abc";

/* wide character string literal, of type wchar_t[] */
wchar_t* s2 = L"abc";
```

Version ≥ C11

```
/* UTF-8 string literal, of type char[] */
char* s3 = u8"abc";

/* 16-bit wide string literal, of type char16_t[] */
char16_t* s4 = u"abc";

/* 32-bit wide string literal, of type char32_t[] */
char32_t* s5 = U"abc";
```

Chapter 4: Operators

An operator in a programming language is a symbol that tells the compiler or interpreter to perform a specific mathematical, relational or logical operation and produce a final result.

C has many powerful operators. Many C operators are binary operators, which means they have two operands. For example, in `a / b`, `/` is a binary operator that accepts two operands (`a`, `b`). There are some unary operators which take one operand (for example: `~`, `++`), and only one ternary operator `? :`.

Section 4.1: Relational Operators

Relational operators check if a specific relation between two operands is true. The result is evaluated to 1 (which means *true*) or 0 (which means *false*). This result is often used to affect control flow (via `if`, `while`, `for`), but can also be stored in variables.

Equals "=="

Checks whether the supplied operands are equal.

```
1 == 0;         /* evaluates to 0. */
1 == 1;         /* evaluates to 1. */

int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr == yptr;   /* evaluates to 0, the operands hold different location addresses. */
*xptr == *yptr; /* evaluates to 1, the operands point at locations that hold the same value. */
```

Attention: This operator should not be confused with the assignment operator (`=`)!

Not equals "!="

Checks whether the supplied operands are not equal.

```
1 != 0;         /* evaluates to 1. */
1 != 1;         /* evaluates to 0. */

int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr != yptr;   /* evaluates to 1, the operands hold different location addresses. */
*xptr != *yptr; /* evaluates to 0, the operands point at locations that hold the same value. */
```

This operator effectively returns the opposite result to that of the equals (`==`) operator.

Not "!"

Check whether an object is equal to `0`.

The `!` can also be used directly with a variable as follows:

```
!someVal
```

This has the same effect as:

```c
someVal == 0
```

Greater than ">"

Checks whether the left hand operand has a greater value than the right hand operand

```c
5 > 4       /* evaluates to 1. */
4 > 5       /* evaluates to 0. */
4 > 4       /* evaluates to 0. */
```

Less than "<"

Checks whether the left hand operand has a smaller value than the right hand operand

```c
5 < 4       /* evaluates to 0. */
4 < 5       /* evaluates to 1. */
4 < 4       /* evaluates to 0. */
```

Greater than or equal ">="

Checks whether the left hand operand has a greater or equal value to the right operand.

```c
5 >= 4      /* evaluates to 1. */
4 >= 5      /* evaluates to 0. */
4 >= 4      /* evaluates to 1. */
```

Less than or equal "<="

Checks whether the left hand operand has a smaller or equal value to the right operand.

```c
5 <= 4      /* evaluates to 0. */
4 <= 5      /* evaluates to 1. */
4 <= 4      /* evaluates to 1. */
```

Section 4.2: Conditional Operator/Ternary Operator

Evaluates its first operand, and, if the resulting value is not equal to zero, evaluates its second operand. Otherwise, it evaluates its third operand, as shown in the following example:

```c
a = b ? c : d;
```

is equivalent to:

```c
if (b)
    a = c;
else
    a = d;
```

This pseudo-code represents it : `condition ? value_if_true : value_if_false`. Each value can be the result of an evaluated expression.

```c
int x = 5;
int y = 42;
printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /* Outputs "5, 42" */
```

The conditional operator can be nested. For example, the following code determines the bigger of three numbers:

```
big= a > b ? (a > c ? a : c)
           : (b > c ? b : c);
```

The following example writes even integers to one file and odd integers to another file:

```
#include<stdio.h>

int main()
{
    FILE *even, *odds;
    int n = 10;
    size_t k = 0;

    even = fopen("even.txt", "w");
    odds = fopen("odds.txt", "w");

    for(k = 1; k < n + 1; k++)
    {
        k%2==0 ? fprintf(even, "\t%5d\n", k)
               : fprintf(odds, "\t%5d\n", k);
    }
    fclose(even);
    fclose(odds);

    return 0;
}
```

The conditional operator associates from right to left. Consider the following:

```
exp1 ? exp2 : exp3 ? exp4 : exp5
```

As the association is from right to left, the above expression is evaluated as

```
exp1 ? exp2 : ( exp3 ? exp4 : exp5 )
```

Section 4.3: Bitwise Operators

Bitwise operators can be used to perform bit level operation on variables.
Below is a list of all six bitwise operators supported in C:

Symbol	Operator
&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR (XOR)
~	bitwise not (one's complement)
<<	logical left shift
>>	logical right shift

Following program illustrates the use of all bitwise operators:

```
#include <stdio.h>

int main(void)
{
```

```c
    unsigned int a = 29;    /* 29 = 0001 1101 */
    unsigned int b = 48;    /* 48 = 0011 0000 */
    int c = 0;

    c = a & b;              /* 32 = 0001 0000 */
    printf("%d & %d = %d\n", a, b, c );

    c = a | b;              /* 61 = 0011 1101 */
    printf("%d | %d = %d\n", a, b, c );

    c = a ^ b;              /* 45 = 0010 1101 */
    printf("%d ^ %d = %d\n", a, b, c );

    c = ~a;                 /* -30 = 1110 0010 */
    printf("~%d = %d\n", a, c );

    c = a << 2;             /* 116 = 0111 0100 */
    printf("%d << 2 = %d\n", a, c );

    c = a >> 2;             /* 7 = 0000 0111 */
    printf("%d >> 2 = %d\n", a, c );

    return 0;
}
```

Bitwise operations with signed types should be avoided because the sign bit of such a bit representation has a particular meaning. Particular restrictions apply to the shift operators:

- Left shifting a 1 bit into the signed bit is erroneous and leads to undefined behavior.
- Right shifting a negative value (with sign bit 1) is implementation defined and therefore not portable.
- If the value of the right operand of a shift operator is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

Masking:

Masking refers to the process of extracting the desired bits from (or transforming the desired bits in) a variable by using logical bitwise operations. The operand (a constant or variable) that is used to perform masking is called a *mask*.

Masking is used in many different ways:

- To decide the bit pattern of an integer variable.
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 0s (using bitwise AND)
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 1s (using bitwise OR).
- To copy a portion of a given bit pattern to a new variable, while the remainder of the original bit pattern is inverted within the new variable (using bitwise exclusive OR).

The following function uses a mask to display the bit pattern of a variable:

```c
#include <limits.h>
void bit_pattern(int u)
{
    int i, x, word;
    unsigned mask = 1;
```

```c
    word = CHAR_BIT * sizeof(int);
    mask = mask << (word - 1);    /* shift 1 to the leftmost position */
    for(i = 1; i <= word; i++)
    {
        x = (u & mask) ? 1 : 0;  /* identify the bit */
        printf("%d", x);         /* print bit value */
        mask >>= 1;              /* shift mask to the right by 1 bit */
    }
}
```

Section 4.4: Short circuit behavior of logical operators

Short circuiting is a functionality that skips evaluating parts of a (if/while/...) condition when able. In case of a logical operation on two operands, the first operand is evaluated (to true or false) and if there is a verdict (i.e first operand is false when using &&, first operand is true when using ||) the second operand is not evaluated.

Example:

```c
#include <stdio.h>

int main(void) {
  int a = 20;
  int b = -5;

  /* here 'b == -5' is not evaluated,
     since a 'a != 20' is false. */
  if (a != 20 && b == -5) {
    printf("I won't be printed!\n");
  }

  return 0;
}
```

Check it out yourself:

```c
#include <stdio.h>

int print(int i) {
  printf("print function %d\n", i);
  return i;
}

int main(void) {
  int a = 20;

  /* here 'print(a)' is not called,
     since a 'a != 20' is false. */
  if (a != 20 && print(a)) {
    printf("I won't be printed!\n");
  }

  /* here 'print(a)' is called,
     since a 'a == 20' is true. */
  if (a == 20 && print(a)) {
    printf("I will be printed!\n");
  }

  return 0;
}
```

Output:

```
$ ./a.out
print function 20
I will be printed!
```

Short circuiting is important, when you want to avoid evaluating terms that are (computationally) costly. Moreover, it can heavily affect the flow of your program like in this case: Why does this program print "forked!" 4 times?

Section 4.5: Comma Operator

Evaluates its left operand, discards the resulting value, and then evaluates its rights operand and result yields the value of its rightmost operand.

```
int x = 42, y = 42;
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
```

The comma operator introduces a sequence point between its operands.

Note that the *comma* used in functions calls that separate arguments is NOT the *comma operator*, rather it's called a *separator* which is different from the *comma operator*. Hence, it doesn't have the properties of the *comma operator*.

The above `printf()` call contains both the *comma operator* and the *separator*.

```
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
/*           ^        ^ this is a comma operator */
/*           this is a separator */
```

The comma operator is often used in the initialization section as well as in the updating section of a `for` loop. For example:

```
for(k = 1; k < 10; printf("\%d\\n", k), k += 2);   /*outputs the odd numbers below 9/*

/* outputs sum to first 9 natural numbers */
for(sumk = 1, k = 1; k < 10; k++, sumk += k)
    printf("\%5d\%5d\\n", k, sumk);
```

Section 4.6: Arithmetic Operators

Basic Arithmetic

Return a value that is the result of applying the left hand operand to the right hand operand, using the associated mathematical operation. Normal mathematical rules of commutation apply (i.e. addition and multiplication are commutative, subtraction, division and modulus are not).

Addition Operator

The addition operator (+) is used to add two operands together. Example:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;
```

```
    int c = a + b; /* c now holds the value 12 */

    printf("%d + %d = %d",a,b,c); /* will output "5 + 7 = 12" */

    return 0;
}
```

Subtraction Operator

The subtraction operator (`-`) is used to subtract the second operand from the first. Example:

```
#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 7;

    int c = a - b; /* c now holds the value 3 */

    printf("%d - %d = %d",a,b,c); /* will output "10 - 7 = 3" */

    return 0;
}
```

Multiplication Operator

The multiplication operator (`*`) is used to multiply both operands. Example:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a * b; /* c now holds the value 35 */

    printf("%d * %d = %d",a,b,c); /* will output "5 * 7 = 35" */

    return 0;
}
```

Not to be confused with the `*` dereference operator.

Division Operator

The division operator (`/`) divides the first operand by the second. If both operands of the division are integers, it will return an integer value and discard the remainder (use the modulo operator `%` for calculating and acquiring the remainder).

If one of the operands is a floating point value, the result is an approximation of the fraction.

Example:

```
#include <stdio.h>

int main (void)
```

```
{
    int a = 19 / 2 ; /* a holds value 9   */
    int b = 18 / 2 ; /* b holds value 9   */
    int c = 255 / 2; /* c holds value 127 */
    int d = 44 / 4 ; /* d holds value 11  */
    double e = 19 / 2.0 ; /* e holds value 9.5   */
    double f = 18.0 / 2 ; /* f holds value 9.0   */
    double g = 255 / 2.0; /* g holds value 127.5 */
    double h = 45.0 / 4 ; /* h holds value 11.25 */

    printf("19 / 2 = %d\n", a);    /* Will output "19 / 2 = 9"    */
    printf("18 / 2 = %d\n", b);    /* Will output "18 / 2 = 9"    */
    printf("255 / 2 = %d\n", c);   /* Will output "255 / 2 = 127" */
    printf("44 / 4 = %d\n", d);    /* Will output "44 / 4 = 11"   */
    printf("19 / 2.0 = %g\n", e);  /* Will output "19 / 2.0 = 9.5"    */
    printf("18.0 / 2 = %g\n", f);  /* Will output "18.0 / 2 = 9"      */
    printf("255 / 2.0 = %g\n", g); /* Will output "255 / 2.0 = 127.5" */
    printf("45.0 / 4 = %g\n", h);  /* Will output "45.0 / 4 = 11.25"  */

    return 0;
}
```

Modulo Operator

The modulo operator (%) receives integer operands only, and is used to calculate the remainder after the first operand is divided by the second. Example:

```
#include <stdio.h>

int main (void) {
    int a = 25 % 2;    /* a holds value 1  */
    int b = 24 % 2;    /* b holds value 0  */
    int c = 155 % 5;   /* c holds value 0  */
    int d = 49 % 25;   /* d holds value 24 */

    printf("25 % 2 = %d\n", a);     /* Will output "25 % 2 = 1"    */
    printf("24 % 2 = %d\n", b);     /* Will output "24 % 2 = 0"    */
    printf("155 % 5 = %d\n", c);    /* Will output "155 % 5 = 0"   */
    printf("49 % 25 = %d\n", d);    /* Will output "49 % 25 = 24"  */

    return 0;
}
```

Increment / Decrement Operators

The increment (a++) and decrement (

a--

) operators are different in that they change the value of the variable you apply them to without an assignment operator. You can use increment and decrement operators either before or after the variable. The placement of the operator changes the timing of the incrementation/decrementation of the value to before or after assigning it to the variable. Example:

```
#include <stdio.h>

int main(void)
{
    int a = 1;
    int b = 4;
```

```c
    int c = 1;
    int d = 4;

    a++;
    printf("a = %d\n",a);     /* Will output "a = 2" */
    b--;
    printf("b = %d\n",b);     /* Will output "b = 3" */

    if (++c > 1) { /* c is incremented by 1 before being compared in the condition */
        printf("This will print\n");    /* This is printed */
    } else {
        printf("This will never print\n");    /* This is not printed */
    }

    if (d-- < 4) {  /* d is decremented after being compared */
        printf("This will never print\n");    /* This is not printed */
    } else {
        printf("This will print\n");    /* This is printed */
    }
}
```

As the example for `c` and `d` shows, both operators have two forms, as prefix notation and postfix notation. Both have the same effect in incrementing (`++`) or decrementing (`--`) the variable, but differ by the value they return: prefix operations do the operation first and then return the value, whereas postfix operations first determine the value that is to be returned, and then do the operation.

Because of this potentially counter-intuitive behaviour, the use of increment/decrement operators inside expressions is controversial.

Section 4.7: Access Operators

The member access operators (dot `.` and arrow `->`) are used to access a member of a `struct`.

Member of object

Evaluates into the lvalue denoting the object that is a member of the accessed object.

```c
struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
myObject.x = 42;
myObject.y = 123;

printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Outputs ".x = 42, .y = 123". */
```

Member of pointed-to object

Syntactic sugar for dereferencing followed by member access. Effectively, an expression of the form `x->y` is shorthand for `(*x).y` — but the arrow operator is much clearer, especially if the structure pointers are nested.

```c
struct MyStruct
{
    int x;
    int y;
```

```c
};

struct MyStruct myObject;
struct MyStruct *p = &myObject;

p->x = 42;
p->y = 123;

printf(".x = %i, .y = %i\n", p->x, p->y); /* Outputs ".x = 42, .y = 123". */
printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Also outputs ".x = 42, .y = 123". */
```

Address-of

The unary `&` operator is the address of operator. It evaluates the given expression, where the resulting object must be an lvalue. Then, it evaluates into an object whose type is a pointer to the resulting object's type, and contains the address of the resulting object.

```c
int x = 3;
int *p = &x;
printf("%p = %p\n", (void *)&x, (void *)p); /* Outputs "A = A", for some implementation-defined A.
*/
```

Dereference

The unary `*` operator dereferences a pointer. It evaluates into the lvalue resulting from dereferencing the pointer that results from evaluating the given expression.

```c
int x = 42;
int *p = &x;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 42, *p = 42". */

*p = 123;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 123, *p = 123". */
```

Indexing

Indexing is syntactic sugar for pointer addition followed by dereferencing. Effectively, an expression of the form `a[i]` is equivalent to `*(a + i)` — but the explicit subscript notation is preferred.

```c
int arr[] = { 1, 2, 3, 4, 5 };
printf("arr[2] = %i\n", arr[2]); /* Outputs "arr[2] = 3". */
```

Interchangeability of indexing

Adding a pointer to an integer is a commutative operation (i.e. the order of the operands does not change the result) so `pointer + integer == integer + pointer`.

A consequence of this is that `arr[3]` and `3[arr]` are equivalent.

```c
printf("3[arr] = %i\n", 3[arr]); /* Outputs "3[arr] = 4". */
```

Usage of an expression `3[arr]` instead of `arr[3]` is generally not recommended, as it affects code readability. It tends to be a popular in obfuscated programming contests.

Section 4.8: sizeof Operator

With a type as operand

Evaluates into the size in bytes, of type `size_t`, of objects of the given type. Requires parentheses around the type.

```
printf("%zu\n", sizeof(int)); /* Valid, outputs the size of an int object, which is platform-
dependent. */
printf("%zu\n", sizeof int); /* Invalid, types as arguments need to be surrounded by parentheses! */
```

With an expression as operand

Evaluates into the size in bytes, of type `size_t`, of objects of the type of the given expression. The expression itself is not evaluated. Parentheses are not required; however, because the given expression must be unary, it's considered best practice to always use them.

```
char ch = 'a';
printf("%zu\n", sizeof(ch)); /* Valid, will output the size of a char object, which is always 1 for
all platforms. */
printf("%zu\n", sizeof ch);  /* Valid, will output the size of a char object, which is always 1 for
all platforms. */
```

Section 4.9: Cast Operator

Performs an *explicit* conversion into the given type from the value resulting from evaluating the given expression.

```
int x = 3;
int y = 4;
printf("%f\n", (double)x / y); /* Outputs "0.750000". */
```

Here the value of x is converted to a `double`, the division promotes the value of y to `double`, too, and the result of the division, a `double` is passed to `printf` for printing.

Section 4.10: Function Call Operator

The first operand must be a function pointer (a function designator is also acceptable because it will be converted to a pointer to the function), identifying the function to call, and all other operands, if any, are collectively known as the function call's arguments. Evaluates into the return value resulting from calling the appropriate function with the respective arguments.

```
int myFunction(int x, int y)
{
    return x * 2 + y;
}

int (*fn)(int, int) = &myFunction;
int x = 42;
int y = 123;

printf("(*fn)(%i, %i) = %i\n", x, y, (*fn)(x, y)); /* Outputs "fn(42, 123) = 207". */
printf("fn(%i, %i) = %i\n", x, y, fn(x, y)); /* Another form: you don't need to dereference
explicitly */
```

Section 4.11: Increment / Decrement

The increment and decrement operators exist in *prefix* and *postfix* form.

```
int a = 1;
int b = 1;
int tmp = 0;

tmp = ++a;        /* increments a by one, and returns new value; a == 2, tmp == 2  */
tmp = a++;        /* increments a by one, but returns old value; a == 3, tmp == 2 */
tmp = --b;        /* decrements b by one, and returns new value; b == 0, tmp == 0 */
tmp = b--;        /* decrements b by one, but returns old value; b == -1, tmp == 0 */
```

Note that arithmetic operations do not introduce sequence points, so certain expressions with `++` or `--` operators may introduce undefined behaviour.

Section 4.12: Assignment Operators

Assigns the value of the right-hand operand to the storage location named by the left-hand operand, and returns the value.

```
int x = 5;      /* Variable x holds the value 5. Returns 5. */
char y = 'c';   /* Variable y holds the value 99. Returns 99
                 * (as the character 'c' is represented in the ASCII table with 99).
                 */
float z = 1.5;  /* variable z holds the value 1.5. Returns 1.5. */
char const* s = "foo"; /* Variable s holds the address of the first character of the string 'foo'.
*/
```

Several arithmetical operations have a *compound assignment* operator.

```
a += b  /* equal to: a = a + b */
a -= b  /* equal to: a = a - b */
a *= b  /* equal to: a = a * b */
a /= b  /* equal to: a = a / b */
a %= b  /* equal to: a = a % b */
a &= b  /* equal to: a = a & b */
a |= b  /* equal to: a = a | b */
a ^= b  /* equal to: a = a ^ b */
a <<= b /* equal to: a = a << b */
a >>= b /* equal to: a = a >> b */
```

One important feature of these compound assignments is that the expression on the left hand side (a) is only evaluated once. E.g if `p` is a pointer

```
*p += 27;
```

dereferences `p` only once, whereas the following does so twice.

```
*p = *p + 27;
```

It should also be noted that the result of an assignment such as `a = b` is what is known as an *rvalue*. Thus, the assignment actually has a value which can then be assigned to another variable. This allows the chaining of assignments to set multiple variables in a single statement.

This *rvalue* can be used in the controlling expressions of `if` statements (or loops or `switch` statements) that guard

some code on the result of another expression or function call. For example:

```
char *buffer;
if ((buffer = malloc(1024)) != NULL)
{
    /* do something with buffer */
    free(buffer);
}
else
{
    /* report allocation failure */
}
```

Because of this, care must be taken to avoid a common typo which can lead to mysterious bugs.

```
int a = 2;
/* ... */
if (a = 1)
    /* Delete all files on my hard drive */
```

This will have disastrous results, as `a = 1` will always evaluate to `1` and thus the controlling expression of the `if` statement will always be true (read more about this common pitfall here). The author almost certainly meant to use the equality operator (`==`) as shown below:

```
int a = 2;
/* ... */
if (a == 1)
    /* Delete all files on my hard drive */
```

Operator Associativity

```
int a, b = 1, c = 2;
a = b = c;
```

This assigns `c` to `b`, which returns `b`, which is than assigned to `a`. This happens because all assignment-operators have right associativity, that means the rightmost operation in the expression is evaluated first, and proceeds from right to left.

Section 4.13: Logical Operators

Logical AND

Performs a logical boolean AND-ing of the two operands returning 1 if both of the operands are non-zero. The logical AND operator is of type `int`.

```
0 && 0  /* Returns 0. */
0 && 1  /* Returns 0. */
2 && 0  /* Returns 0. */
2 && 3  /* Returns 1. */
```

Logical OR

Performs a logical boolean OR-ing of the two operands returning 1 if any of the operands are non-zero. The logical OR operator is of type `int`.

```
0 || 0  /* Returns 0. */
```

```
0 || 1  /* Returns 1.  */
2 || 0  /* Returns 1.  */
2 || 3  /* Returns 1.  */
```

Logical NOT

Performs a logical negation. The logical NOT operator is of type `int`. The NOT operator checks if at least one bit is equal to 1, if so it returns 0. Else it returns 1;

```
!1 /* Returns 0. */
!5 /* Returns 0. */
!0 /* Returns 1. */
```

Short-Circuit Evaluation

There are some crucial properties common to both `&&` and `||`:

- the left-hand operand (LHS) is fully evaluated before the right-hand operand (RHS) is evaluated at all,
- there is a sequence point between the evaluation of the left-hand operand and the right-hand operand,
- and, most importantly, the right-hand operand is not evaluated at all if the result of the left-hand operand determines the overall result.

This means that:

- if the LHS evaluates to 'true' (non-zero), the RHS of `||` will not be evaluated (because the result of 'true OR anything' is 'true'),
- if the LHS evaluates to 'false' (zero), the RHS of `&&` will not be evaluated (because the result of 'false AND anything' is 'false').

This is important as it permits you to write code such as:

```
const char *name_for_value(int value)
{
    static const char *names[] = { "zero", "one", "two", "three", };
    enum { NUM_NAMES = sizeof(names) / sizeof(names[0]) };
    return (value >= 0 && value < NUM_NAMES) ? names[value] : "infinity";
}
```

If a negative value is passed to the function, the `value >= 0` term evaluates to false and the `value < NUM_NAMES` term is not evaluated.

Section 4.14: Pointer Arithmetic

Pointer addition

Given a pointer and a scalar type `N`, evaluates into a pointer to the `N`th element of the pointed-to type that directly succeeds the pointed-to object in memory.

```
int arr[] = {1, 2, 3, 4, 5};
printf("*(arr + 3) = %i\n", *(arr + 3)); /* Outputs "4", arr's fourth element. */
```

It does not matter if the pointer is used as the operand value or the scalar value. This means that things such as `3 + arr` are valid. If `arr[k]` is the `k+1` member of an array, then `arr+k` is a pointer to `arr[k]`. In other words, `arr` or `arr+0` is a pointer to `arr[0]`, `arr+1` is a pointer to `arr[2]`, and so on. In general, `*(arr+k)` is same as `arr[k]`.

Unlike the usual arithmetic, addition of `1` to a pointer to an `int` will add `4` bytes to the current address value. As array names are constant pointers, `+` is the only operator we can use to access the members of an array via pointer notation using the array name. However, by defining a pointer to an array, we can get more flexibility to process the data in an array. For example, we can print the members of an array as follows:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", *(arr + k));
    }
    return 0;
}
```

By defining a pointer to the array, the above program is equivalent to the following:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr; /* or int *ptr = &arr[0]; */
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", ptr[k]);
        /* or   printf("\n\t%d", *(ptr + k)); */
        /* or   printf("\n\t%d", *ptr++); */
    }
    return 0;
}
```

See that the members of the array `arr` are accessed using the operators `+` and `++`. The other operators that can be used with the pointer `ptr` are `-` and `--`.

Pointer subtraction

Given two pointers to the same type, evaluates into an object of type `ptrdiff_t` that holds the scalar value that must be added to the second pointer in order to obtain the value of the first pointer.

```
int arr[] = {1, 2, 3, 4, 5};
int *p = &arr[2];
int *q = &arr[3];
ptrdiff_t diff = q - p;

printf("q - p = %ti\n", diff); /* Outputs "1". */
printf("*(p + (q - p)) = %d\n", *(p + diff)); /* Outputs "4". */
```

Section 4.15: _Alignof

Version ≥ C11

Queries the alignment requirement for the specified type. The alignment requirement is a positive integral power of 2 representing the number of bytes between which two objects of the type may be allocated. In C, the alignment requirement is measured in `size_t`.

The type name may not be an incomplete type nor a function type. If an array is used as the type, the type of the array element is used.

This operator is often accessed through the convenience macro `alignof` from **`<stdalign.h>`**.

```
int main(void)
{
    printf("Alignment of char = %zu\n", alignof(char));
    printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
    printf("alignof(float[10]) = %zu\n", alignof(float[10]));
    printf("alignof(struct{char c; int n;}) = %zu\n",
            alignof(struct {char c; int n;}));
}
```

Possible Output:

```
Alignment of char = 1
Alignment of max_align_t = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

http://en.cppreference.com/w/c/language/_Alignof

Chapter 5: Boolean

Section 5.1: Using stdbool.h

Version ≥ C99

Using the system header file `stdbool.h` allows you to use `bool` as a Boolean data type. **`true`** evaluates to `1` and **`false`** evaluates to `0`.

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true;  /* equivalent to bool x = 1; */
    bool y = false; /* equivalent to bool y = 0; */
    if (x)  /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

`bool` is just a nice spelling for the data type `_Bool`. It has special rules when numbers or pointers are converted to it.

Section 5.2: Using #define

C of all versions, will effectively treat any integer value other than `0` as **`true`** for comparison operators and the integer value `0` as **`false`**. If you don't have `_Bool` or `bool` as of C99 available, you could simulate a Boolean data type in C using `#define` macros, and you might still find such things in legacy code.

```
#include <stdio.h>

#define bool int
#define true 1
#define false 0

int main(void) {
    bool x = true;  /* Equivalent to int x = 1; */
    bool y = false; /* Equivalent to int y = 0; */
    if (x) /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

Don't introduce this in new code since the definition of these macros might clash with modern uses of **`<stdbool.h>`**.

Section 5.3: Using the Intrinsic (built-in) Type _Bool

Version ≥ C99

Added in the C standard version C99, `_Bool` is also a native C data type. It is capable of holding the values `0` (for *false*) and `1` (for *true*).

```
#include <stdio.h>

int main(void) {
    _Bool x = 1;
    _Bool y = 0;
    if(x) /* Equivalent to if (x == 1) */
    {
        puts("This will print!");
    }
    if (!y) /* Equivalent to if (y == 0) */
    {
        puts("This will also print!");
    }
}
```

`_Bool` is an integer type but has special rules for conversions from other types. The result is analogous to the usage of other types in `if` expressions. In the following

```
_Bool z = X;
```

- If `X` has an arithmetic type (is any kind of number), `z` becomes `0` if `X == 0`. Otherwise `z` becomes `1`.
- If `X` has a pointer type, `z` becomes `0` if `X` is a null pointer and `1` otherwise.

To use nicer spellings `bool`, **`false`** and **`true`** you need to use **`<stdbool.h>`**.

Section 5.4: Integers and pointers in Boolean expressions

All integers or pointers can be used in an expression that is interpreted as "truth value".

```
int main(int argc, char* argv[]) {
  if (argc % 4) {
    puts("arguments number is not divisible by 4");
  } else {
    puts("argument number is divisible by 4");
  }
...
```

The expression `argc % 4` is evaluated and leads to one of the values `0`, `1`, `2` or `3`. The first, `0` is the only value that is "false" and brings execution into the `else` part. All other values are "true" and go into the `if` part.

```
double* A = malloc(n*sizeof *A);
if (!A) {
   perror("allocation problems");
   exit(EXIT_FAILURE);
}
```

Here the pointer `A` is evaluated and if it is a null pointer, an error is detected and the program exits.

Many people prefer to write something as `A == NULL`, instead, but if you have such pointer comparisons as part of

other complicated expressions, things become quickly difficult to read.

```
char const* s = ....;   /* some pointer that we receive */
if (s != NULL && s[0] != '\0' && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

For this to check, you'd have to scan a complicated code in the expression and be sure about operator preference.

```
char const* s = ....;   /* some pointer that we receive */
if (s && s[0] && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

is relatively easy to capture: if the pointer is valid we check if the first character is non-zero and then check if it is a letter.

Section 5.5: Defining a bool type using typedef

Considering that most debuggers are not aware of `#define` macros, but can check **enum** constants, it may be desirable to do something like this:

```
#if __STDC_VERSION__ < 199900L
typedef enum { false, true } bool;
/* Modern C code might expect these to be macros. */
# ifndef bool
#  define bool bool
# endif
# ifndef true
#  define true true
# endif
# ifndef false
#  define false false
# endif
#else
# include <stdbool.h>
#endif

/* Somewhere later in the code ... */
bool b = true;
```

This allows compilers for historic versions of C to function, but remains forward compatible if the code is compiled with a modern C compiler.

For more information on `typedef`, see Typedef, for more on **enum** see Enumerations

Chapter 6: Strings

In C, a string is not an intrinsic type. A C-string is the convention to have a one-dimensional array of characters which is terminated by a null-character, by a `'\0'`.

This means that a C-string with a content of `"abc"` will have four characters `'a'`, `'b'`, `'c'` and `'\0'`.

See the basic introduction to strings example.

Section 6.1: Tokenisation: strtok(), strtok_r() and strtok_s()

The function `strtok` breaks a string into a smaller strings, or tokens, using a set of delimiters.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int toknum = 0;
    char src[] = "Hello,, world!";
    const char delimiters[] = ", !";
    char *token = strtok(src, delimiters);
    while (token != NULL)
    {
        printf("%d: [%s]\n", ++toknum, token);
        token = strtok(NULL, delimiters);
    }
    /* source is now "Hello\0, world\0\0" */
}
```

Output:

```
1: [Hello]
2: [world]
```

The string of delimiters may contain one or more delimiters and different delimiter strings may be used with each call to `strtok`.

Calls to `strtok` to continue tokenizing the same source string should not pass the source string again, but instead pass `NULL` as the first argument. If the same source string *is* passed then the first token will instead be re-tokenized. That is, given the same delimiters, `strtok` would simply return the first token again.

Note that as `strtok` does not allocate new memory for the tokens, *it modifies the source string*. That is, in the above example, the string `src` will be manipulated to produce the tokens that are referenced by the pointer returned by the calls to `strtok`. This means that the source string cannot be `const` (so it can't be a string literal). It also means that the identity of the delimiting byte is lost (i.e. in the example the "," and "!" are effectively deleted from the source string and you cannot tell which delimiter character matched).

Note also that multiple consecutive delimiters in the source string are treated as one; in the example, the second comma is ignored.

`strtok` is neither thread safe nor re-entrant because it uses a static buffer while parsing. This means that if a function calls `strtok`, no function that it calls while it is using `strtok` can also use `strtok`, and it cannot be called by any function that is itself using `strtok`.

An example that demonstrates the problems caused by the fact that `strtok`is not re-entrant is as follows:

```
char src[] = "1.2,3.5,4.2";
char *first = strtok(src, ",");

do
{
    char *part;
    /* Nested calls to strtok do not work as desired */
    printf("[%s]\n", first);
    part = strtok(first, ".");
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok(NULL, ".");
    }
} while ((first = strtok(NULL, ",")) != NULL);
```

Output:

```
[1.2]
[1]
[2]
```

The expected operation is that the outer `do while` loop should create three tokens consisting of each decimal number string (`"1.2"`, `"3.5"`, `"4.2"`), for each of which the `strtok` calls for the inner loop should split it into separate digit strings (`"1"`, `"2"`, `"3"`, `"5"`, `"4"`, `"2"`).

However, because `strtok` is not re-entrant, this does not occur. Instead the first `strtok` correctly creates the "1.2\0" token, and the inner loop correctly creates the tokens `"1"` and `"2"`. But then the `strtok` in the outer loop is at the end of the string used by the inner loop, and returns NULL immediately. The second and third substrings of the `src` array are not analyzed at all.

Version < C11

The standard C libraries do not contain a thread-safe or re-entrant version but some others do, such as POSIX' `strtok_r`. Note that on MSVC the `strtok` equivalent, `strtok_s` is thread-safe.

Version ≥ C11

C11 has an optional part, Annex K, that offers a thread-safe and re-entrant version named `strtok_s`. You can test for the feature with `__STDC_LIB_EXT1__`. This optional part is not widely supported.

The `strtok_s` function differs from the POSIX `strtok_r` function by guarding against storing outside of the string being tokenized, and by checking runtime constraints. On correctly written programs, though, the `strtok_s` and `strtok_r` behave the same.

Using `strtok_s` with the example now yields the correct response, like so:

```
/* you have to announce that you want to use Annex K */
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

#ifndef __STDC_LIB_EXT1__
# error "we need strtok_s from Annex K"
#endif
```

```c
char src[] = "1.2,3.5,4.2";
char *next = NULL;
char *first = strtok_s(src, ",", &next);

do
{
    char *part;
    char *posn;

    printf("[%s]\n", first);
    part = strtok_s(first, ".", &posn);
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok_s(NULL, ".", &posn);
    }
}
while ((first = strtok_s(NULL, ",", &next)) != NULL);
```

And the output will be:

```
[1.2]
[1]
[2]
[3.5]
[3]
[5]
[4.2]
[4]
[2]
```

Section 6.2: String literals

String literals represent null-terminated, static-duration arrays of `char`. Because they have static storage duration, a string literal or a pointer to the same underlying array can safely be used in several ways that a pointer to an automatic array cannot. For example, returning a string literal from a function has well-defined behavior:

```c
const char *get_hello() {
    return "Hello, World!";  /* safe */
}
```

For historical reasons, the elements of the array corresponding to a string literal are not formally `const`. Nevertheless, any attempt to modify them has undefined behavior. Typically, a program that attempts to modify the array corresponding to a string literal will crash or otherwise malfunction.

```c
char *foo = "hello";
foo[0] = 'y';  /* Undefined behavior - BAD! */
```

Where a pointer points to a string literal -- or where it sometimes may do -- it is advisable to declare that pointer's referent `const` to avoid engaging such undefined behavior accidentally.

```c
const char *foo = "hello";
/* GOOD: can't modify the string pointed to by foo */
```

On the other hand, a pointer to or into the underlying array of a string literal is not itself inherently special; its value can freely be modified to point to something else:

```
char *foo = "hello";
foo = "World!"; /* OK - we're just changing what foo points to */
```

Furthermore, although initializers for `char` arrays can have the same form as string literals, use of such an initializer does not confer the characteristics of a string literal on the initialized array. The initializer simply designates the length and initial contents of the array. In particular, the elements are modifiable if not explicitly declared `const`:

```
char foo[] = "hello";
foo[0] = 'y';  /* OK! */
```

Section 6.3: Calculate the Length: strlen()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        puts("Argument missing.");
        return EXIT_FAILURE;
    }

    size_t len = strlen(argv[1]);
    printf("The length of the second argument is %zu.\\n", len);

    return EXIT_SUCCESS;
}
```

This program computes the length of its second input argument and stores the result in `len`. It then prints that length to the terminal. For example, when run with the parameters `program_name "Hello, world!"`, the program will output `The length of the second argument is 13.` because the string `Hello, world!` is 13 characters long.

`strlen` counts all the **bytes** from the beginning of the string up to, but not including, the terminating NUL character, `'\\0'`. As such, it can only be used when the string is *guaranteed* to be NUL-terminated.

Also keep in mind that if the string contains any Unicode characters, `strlen` will not tell you how many characters are in the string (since some characters may be multiple bytes long). In such cases, you need to count the characters (*i.e.*, code units) yourself. Consider the output of the following example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char asciiString[50] = "Hello world!";
    char utf8String[50] = "Γειά σου Κόσμε!"; /* "Hello World!" in Greek */

    printf("asciiString has %zu bytes in the array\\n", sizeof(asciiString));
    printf("utf8String has %zu bytes in the array\\n", sizeof(utf8String));
    printf("\\"%s\\" is %zu bytes\\n", asciiString, strlen(asciiString));
    printf("\\"%s\\" is %zu bytes\\n", utf8String, strlen(utf8String));
}
```

Output:

```
asciiString has 50 bytes in the array
utf8String has 50 bytes in the array
"Hello world!" is 12 bytes
"Γειά σου Κόσμε!" is 27 bytes
```

Section 6.4: Basic introduction to strings

In C, a **string** is a sequence of characters that is terminated by a null character ('\0').

We can create strings using **string literals**, which are sequences of characters surrounded by double quotation marks; for example, take the string literal `"hello world"`. String literals are automatically null-terminated.

We can create strings using several methods. For instance, we can declare a `char *` and initialize it to point to the first character of a string:

```
char * string = "hello world";
```

When initializing a `char *` to a string constant as above, the string itself is usually allocated in read-only data; `string` is a pointer to the first element of the array, which is the character `'h'`.

Since the string literal is allocated in read-only memory, it is non-modifiable1. Any attempt to modify it will lead to undefined behaviour, so it's better to add `const` to get a compile-time error like this

```
char const * string = "hello world";
```

It has similar effect2 as

```
char const string_arr[] = "hello world";
```

To create a modifiable string, you can declare a character array and initialize its contents using a string literal, like so:

```
char modifiable_string[] = "hello world";
```

This is equivalent to the following:

```
char modifiable_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
```

Since the second version uses brace-enclosed initializer, the string is not automatically null-terminated unless a `'\0'` character is included explicitly in the character array usually as its last element.

1 Non-modifiable implies that the characters in the string literal can't be modified, but remember that the pointer `string` can be modified (can point somewhere else or can be incremented or decremented).

2 Both strings have similar effect in a sense that characters of both strings can't be modified. It should be noted that `string` is a pointer to `char` and it is a modifiable l-value so it can be incremented or point to some other location while the array `string_arr` is a non-modifiable l-value, it can't be modified.

Section 6.5: Copying strings

Pointer assignments do not copy strings

You can use the = operator to copy integers, but you cannot use the = operator to copy strings in C. Strings in C are represented as arrays of characters with a terminating null-character, so using the = operator will only save the address (pointer) of a string.

```
#include <stdio.h>

int main(void) {
    int a = 10, b;
    char c[] = "abc", *d;

    b = a; /* Integer is copied */
    a = 20; /* Modifying a leaves b unchanged - b is a 'deep copy' of a */
    printf("%d %d\n", a, b); /* "20 10" will be printed */

    d = c;
    /* Only copies the address of the string -
    there is still only one string stored in memory */

    c[1] = 'x';
    /* Modifies the original string - d[1] = 'x' will do exactly the same thing */

    printf("%s %s\n", c, d); /* "axc axc" will be printed */

    return 0;
}
```

The above example compiled because we used `char *d` rather than `char d[3]`. Using the latter would cause a compiler error. You cannot assign to arrays in C.

```
#include <stdio.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    b = a; /* compile error */
    printf("%s\n", b);

    return 0;
}
```

Copying strings using standard functions

`strcpy()`

To actually copy strings, `strcpy()` function is available in `string.h`. Enough space must be allocated for the destination before copying.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    strcpy(b, a); /* think "b special equals a" */
    printf("%s\n", b); /* "abc" will be printed */

    return 0;
}
```

Version ≥ C99

snprintf()

To avoid buffer overrun, snprintf() may be used. It is not the best solution performance-wise since it has to parse the template string, but it is the only buffer limit-safe function for copying strings readily-available in standard library, that can be used without any extra steps.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "012345678901234567890";
    char b[8];

#if 0
    strcpy(b, a); /* causes buffer overrun (undefined behavior), so do not execute this here! */
#endif

    snprintf(b, sizeof(b), "%s", a); /* does not cause buffer overrun */
    printf("%s\n", b); /* "0123456" will be printed */

    return 0;
}
```

strncat()

A second option, with better performance, is to use strncat() (a buffer overflow checking version of strcat()) - it takes a third argument that tells it the maximum number of bytes to copy:

```
char dest[32];

dest[0] = '\0';
strncat(dest, source, sizeof(dest) - 1);
    /* copies up to the first (sizeof(dest) - 1) elements of source into dest,
    then puts a \0 on the end of dest */
```

Note that this formulation use `sizeof(dest) - 1`; this is crucial because `strncat()` always adds a null byte (good), but doesn't count that in the size of the string (a cause of confusion and buffer overwrites).

Also note that the alternative — concatenating after a non-empty string — is even more fraught. Consider:

```
char dst[24] = "Clownfish: ";
char src[] = "Marvin and Nemo";
size_t len = strlen(dst);

strncat(dst, src, sizeof(dst) - len - 1);
printf("%zu: [%s]\n", strlen(dst), dst);
```

The output is:

```
23: [Clownfish: Marvin and N]
```

Note, though, that the size specified as the length was *not* the size of the destination array, but the amount of space left in it, not counting the terminal null byte. This can cause big overwriting problems. It is also a bit wasteful; to specify the length argument correctly, you know the length of the data in the destination, so you could instead specify the address of the null byte at the end of the existing content, saving `strncat()` from rescanning it:

```
    strcpy(dst, "Clownfish: ");
```

```
    assert(len < sizeof(dst) - 1);
    strncat(dst + len, src, sizeof(dst) - len - 1);
    printf("%zu: [%s]\n", strlen(dst), dst);
```

This produces the same output as before, but `strncat()` doesn't have to scan over the existing content of `dst` before it starts copying.

`strncpy()`

The last option is the `strncpy()` function. Although you might think it should come first, it is a rather deceptive function that has two main gotchas:

1. If copying via `strncpy()` hits the buffer limit, a terminating null-character won't be written.
2. `strncpy()` always completely fills the destination, with null bytes if necessary.

(Such quirky implementation is historical and was initially intended for handling UNIX file names)

The only correct way to use it is to manually ensure null-termination:

```
strncpy(b, a, sizeof(b)); /* the third parameter is destination buffer size */
b[sizeof(b)/sizeof(*b) - 1] = '\0'; /* terminate the string */
printf("%s\n", b); /* "0123456" will be printed */
```

Even then, if you have a big buffer it becomes very inefficient to use `strncpy()` because of additional null padding.

Section 6.6: Iterating Over the Characters in a String

If we know the length of the string, we can use a for loop to iterate over its characters:

```
char * string = "hello world"; /* This 11 chars long, excluding the 0-terminator. */
size_t i = 0;
for (; i < 11; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Alternatively, we can use the standard function `strlen()` to get the length of a string if we don't know what the string is:

```
size_t length = strlen(string);
size_t i = 0;
for (; i < length; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Finally, we can take advantage of the fact that strings in C are guaranteed to be null-terminated (which we already did when passing it to `strlen()` in the previous example ;-)). We can iterate over the array regardless of its size and stop iterating once we reach a null-character:

```
size_t i = 0;
while (string[i] != '\0') {       /* Stop looping when we reach the null-character. */
    printf("%c\n", string[i]);    /* Print each character of the string. */
    i++;
}
```

Section 6.7: Creating Arrays of Strings

An array of strings can mean a couple of things:

1. An array whose elements are `char *`s
2. An array whose elements are arrays of `char`s

We can create an array of character pointers like so:

```
char * string_array[] = {
    "foo",
    "bar",
    "baz"
};
```

Remember: when we assign string literals to `char *`, the strings themselves are allocated in read-only memory. However, the array `string_array` is allocated in read/write memory. This means that we can modify the pointers in the array, but we cannot modify the strings they point to.

In C, the parameter to main `argv` (the array of command-line arguments passed when the program was run) is an array of `char *`: `char * argv[]`.

We can also create arrays of character arrays. Since strings are arrays of characters, an array of strings is simply an array whose elements are arrays of characters:

```
char modifiable_string_array_literals[][4] = {
    "foo",
    "bar",
    "baz"
};
```

This is equivalent to:

```
char modifiable_string_array[][4] = {
    {'f', 'o', 'o', '\0'},
    {'b', 'a', 'r', '\0'},
    {'b', 'a', 'z', '\0'}
};
```

Note that we specify `4` as the size of the second dimension of the array; each of the strings in our array is actually 4 bytes since we must include the null-terminating character.

Section 6.8: Convert Strings to Number: atoi(), atof() (dangerous, don't use them)

Warning: The functions `atoi`, `atol`, `atoll` and `atof` are inherently unsafe, because: *If the value of the result cannot be represented, the behavior is undefined.* (7.20.1p1)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int val;
    if (argc < 2)
    {
```

```
        printf("Usage: %s <integer>\n", argv[0]);
        return 0;
    }

    val = atoi(argv[1]);

    printf("String value = %s, Int value = %d\n", argv[1], val);

    return 0;
}
```

When the string to be converted is a valid decimal integer that is in range, the function works:

```
$ ./atoi 100
String value = 100, Int value = 100
$ ./atoi 200
String value = 200, Int value = 200
```

For strings that start with a number, followed by something else, only the initial number is parsed:

```
$ ./atoi 0x200
0
$ ./atoi 0123x300
123
```

In all other cases, the behavior is undefined:

```
$ ./atoi hello
Formatting the hard disk...
```

Because of the ambiguities above and this undefined behavior, the `atoi` family of functions should never be used.

- To convert to `long int`, use `strtol()` instead of `atol()`.
- To convert to `double`, use `strtod()` instead of `atof()`.

Version ≥ C99

- To convert to `long long int`, use `strtoll()` instead of `atoll()`.

Section 6.9: string formatted data read/write

Write formatted data to string

```
int sprintf ( char * str, const char * format, ... );
```

use `sprintf` function to write float data to string.

```
#include <stdio.h>
int main ()
{
  char buffer [50];
  double PI = 3.1415926;
  sprintf (buffer, "PI = %.7f", PI);
  printf ("%s\n",buffer);
  return 0;
}
```

Read formatted data from string

```
int sscanf ( const char * s, const char * format, ...);
```

use `sscanf`function to parse formatted data.

```
#include <stdio.h>
int main ()
{
  char sentence []="date : 06-06-2012";
  char str [50];
  int year;
  int month;
  int day;
  sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);
  printf ("%s -> %02d-%02d-%4d\n",str, day, month, year);
  return 0;
}
```

Section 6.10: Find first/last occurrence of a specific character: strchr(), strrchr()

The `strchr` and `strrchr` functions find a character in a string, that is in a NUL-terminated character array. `strchr` return a pointer to the first occurrence and `strrchr` to the last one.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char toSearchFor = 'A';

    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        printf("Argument missing.\n");
        return EXIT_FAILURE;
    }

    {
        char *firstOcc = strchr(argv[1], toSearchFor);
        if (firstOcc != NULL)
        {
            printf("First position of %c in %s is %td.\n",
              toSearchFor, argv[1], firstOcc-argv[1]); /* A pointer difference's result
                                     is a signed integer and uses the length modifier 't'. */
        }
        else
        {
            printf("%c is not in %s.\n", toSearchFor, argv[1]);
        }
    }

    {
        char *lastOcc = strrchr(argv[1], toSearchFor);
        if (lastOcc != NULL)
        {
            printf("Last position of %c in %s is %td.\n",
```

```
                toSearchFor, argv[1], lastOcc-argv[1]);
        }
    }

    return EXIT_SUCCESS;
}
```

Outputs (after having generate an executable named `pos`):

```
$ ./pos AAAAAAA
First position of A in AAAAAAA is 0.
Last position of A in AAAAAAA is 6.
$ ./pos BAbbbbbAccccAAAAzzz
First position of A in BAbbbbbAccccAAAAzzz is 1.
Last position of A in BAbbbbbAccccAAAAzzz is 15.
$  ./pos qwerty
A is not in qwerty.
```

One common use for `strrchr` is to extract a file name from a path. For example to extract `myfile.txt` from `C:\Users\eak\myfile.txt`:

```
char *getFileName(const char *path)
{
    char *pend;

    if ((pend = strrchr(path, '\')) != NULL)
        return pend + 1;

    return NULL;
}
```

Section 6.11: Copy and Concatenation: strcpy(), strcat()

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  /* Always ensure that your string is large enough to contain the characters
   * and a terminating NUL character ('\0')!
   */
  char mystring[10];

  /* Copy "foo" into `mystring`, until a NUL character is encountered. */
  strcpy(mystring, "foo");
  printf("%s\n", mystring);

  /* At this point, we used 4 chars of `mystring`, the 3 characters of "foo",
   * and the NUL terminating byte.
   */

  /* Append "bar" to `mystring`. */
  strcat(mystring, "bar");
  printf("%s\n", mystring);

  /* We now use 7 characters of `mystring`: "foo" requires 3, "bar" requires 3
   * and there is a terminating NUL character ('\0') at the end.
   */
```

```c
  /* Copy "bar" into `mystring`, overwriting the former contents. */
  strcpy(mystring, "bar");
  printf("%s\n", mystring);

  return 0;
}
```

Outputs:

```
foo
foobar
bar
```

If you append to or from or copy from an existing string, ensure it is NUL-terminated!

String literals (e.g. `"foo"`) will always be NUL-terminated by the compiler.

Section 6.12: Comparsion: strcmp(), strncmp(), strcasecmp(), strncasecmp()

The `strcase*`-functions are not Standard C, but a POSIX extension.

The `strcmp` function lexicographically compare two null-terminated character arrays. The functions return a negative value if the first argument appears before the second in lexicographical order, zero if they compare equal, or positive if the first argument appears after the second in lexicographical order.

```c
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcmp(lhs, rhs); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "BBB");
    compare("BBB", "CCCCC");
    compare("BBB", "AAAAAA");
    return 0;
}
```

Outputs:

```
BBB equals BBB
BBB comes before CCCCC
BBB comes after AAAAAA
```

As `strcmp`, `strcasecmp` function also compares lexicographically its arguments after translating each character to its lowercase correspondent:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcasecmp(lhs, rhs); // compute case-insensitive comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "bBB");
    compare("BBB", "ccCCC");
    compare("BBB", "aaaaaa");
    return 0;
}
```

Outputs:

```
BBB equals bBB
BBB comes before ccCCC
BBB comes after aaaaaa
```

`strncmp` and `strncasecmp` compare at most n characters:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs, int n)
{
    int result = strncmp(lhs, rhs, n); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "Bb", 1);
    compare("BBB", "Bb", 2);
    compare("BBB", "Bb", 3);
    return 0;
}
```

Outputs:

```
BBB equals Bb
BBB comes before Bb
BBB comes before Bb
```

Section 6.13: Safely convert Strings to Number: strtoX functions

Version ≥ C99

Since C99 the C library has a set of safe conversion functions that interpret a string as a number. Their names are of the form `strtoX`, where `X` is one of `l`, `ul`, `d`, etc to determine the target type of the conversion

```
double strtod(char const* p, char** endptr);
long double strtold(char const* p, char** endptr);
```

They provide checking that a conversion had an over- or underflow:

```
double ret = strtod(argv[1], 0); /* attempt conversion */

/* check the conversion result. */
if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return;  /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

If the string in fact contains no number at all, this usage of `strtod` returns `0.0`.

If this is not satisfactory, the additional parameter `endptr` can be used. It is a pointer to pointer that will be pointed to the end of the detected number in the string. If it is set to `0`, as above, or `NULL`, it is simply ignored.

This `endptr` parameter provides indicates if there has been a successful conversion and if so, where the number ended:

```
char *check = 0;
double ret = strtod(argv[1], &check); /* attempt conversion */

/* check the conversion result. */
if (argv[1] == check)
    return; /* No number was detected in string */
else if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return; /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

There are analogous functions to convert to the wider integer types:

```
long strtol(char const* p, char** endptr, int nbase);
long long strtoll(char const* p, char** endptr, int nbase);
unsigned long strtoul(char const* p, char** endptr, int nbase);
unsigned long long strtoull(char const* p, char** endptr, int nbase);
```

These functions have a third parameter `nbase` that holds the number base in which the number is written.

```
long a = strtol("101",   0, 2 ); /* a = 5L */
long b = strtol("101",   0, 8 ); /* b = 65L */
long c = strtol("101",   0, 10); /* c = 101L */
long d = strtol("101",   0, 16); /* d = 257L */
long e = strtol("101",   0, 0 ); /* e = 101L */
```

```c
long f = strtol("0101",  0, 0 ); /* f = 65L */
long g = strtol("0x101", 0, 0 ); /* g = 257L */
```

The special value `0` for `nbase` means the string is interpreted in the same way as number literals are interpreted in a C program: a prefix of `0x` corresponds to a hexadecimal representation, otherwise a leading `0` is octal and all other numbers are seen as decimal.

Thus the most practical way to interpret a command-line argument as a number would be

```c
int main(int argc, char* argv[] {
    if (argc < 1)
        return EXIT_FAILURE; /* No number given. */

    /* use strtoull because size_t may be wide */
    size_t mySize = strtoull(argv[1], 0, 0);

    /* then check conversion results. */

     ...

    return EXIT_SUCCESS;
}
```

This means that the program can be called with a parameter in octal, decimal or hexadecimal.

Section 6.14: strspn and strcspn

Given a string, `strspn` calculates the length of the initial substring (span) consisting solely of a specific list of characters. `strcspn` is similar, except it calculates the length of the initial substring consisting of any characters except those listed:

```c
/*
  Provided a string of "tokens" delimited by "separators", print the tokens along
  with the token separators that get skipped.
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    const char sepchars[] = ",.;!?";
    char foo[] = ";ball call,.fall gall hall!?.,";
    char *s;
    int n;

    for (s = foo; *s != 0; /*empty*/) {
        /* Get the number of token separator characters. */
        n = (int)strspn(s, sepchars);

        if (n > 0)
            printf("skipping separators: << %.*s >> (length=%d)\n", n, s, n);

        /* Actually skip the separators now. */
        s += n;

        /* Get the number of token (non-separator) characters. */
        n = (int)strcspn(s, sepchars);
```

```
        if (n > 0)
            printf("token found: << %.*s >> (length=%d)\n", n, s, n);

        /* Skip the token now. */
        s += n;
    }

    printf("== token list exhausted ==\n");

    return 0;
}
```

Analogous functions using wide-character strings are `wcsspn` and `wcscspn`; they're used the same way.

Chapter 7: Literals for numbers, characters and strings

Section 7.1: Floating point literals

Floating point literals are used to represent signed real numbers. The following suffixes can be used to specify type of a literal:

Suffix	Type	Examples
none	`double`	`3.1415926` `-3E6`
`f`, `F`	`float`	`3.1415926f` `2.1E-6F`
`l`, `L`	`long double`	`3.1415926L` `1E126L`

In order to use these suffixes, the literal *must* be a floating point literal. For example, `3f` is an error, since `3` is an integer literal, while `3.f` or `3.0f` are correct. For `long double`, the recommendation is to always use capital `L` for the sake of readability.

Section 7.2: String literals

String literals are used to specify arrays of characters. They are sequences of characters enclosed within double quotes (e.g. `"abcd"` and have the type `char*`).

The `L` prefix makes the literal a wide character array, of type `wchar_t*`. For example, `L"abcd"`.

Since C11, there are other encoding prefixes, similar to `L`:

prefix	base type	encoding
none	`char`	platform dependent
`L`	`wchar_t`	platform dependent
`u8`	`char`	UTF-8
`u`	`char16_t`	usually UTF-16
`U`	`char32_t`	usually UTF-32

For the latter two, it can be queried with feature test macros if the encoding is effectively the corresponding UTF encoding.

Section 7.3: Character literals

Character literals are a special type of integer literals that are used to represent one character. They are enclosed in single quotes, e.g. `'a'` and have the type `int`. The value of the literal is an integer value according to the machine's character set. They do not allow suffixes.

The `L` prefix before a character literal makes it a wide character of type `wchar_t`. Likewise since C11 `u` and `U` prefixes make it wide characters of type `char16_t` and `char32_t`, respectively.

When intending to represent certain special characters, such as a character that is non-printing, escape sequences are used. Escape sequences use a sequence of characters that are translated into another character. All escape sequences consist of two or more characters, the first of which is a backslash `\`. The characters immediately following the backslash determine what character literal the sequence is interpreted as.

Escape Sequence	Represented Character

`\b`	Backspace
`\f`	Form feed
`\n`	Line feed (new line)
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\nnn`	Octal value
`\xnn...`	Hexadecimal value

Version ≥ C89

Escape Sequence	Represented Character
`\a`	Alert (beep, bell)

Version ≥ C99

Escape Sequence	Represented Character
`\unnnn`	Universal character name
`\Unnnnnnnn`	Universal character name

A universal character name is a Unicode code point. A universal character name may map to more than one character. The digits `n` are interpreted as hexadecimal digits. Depending on the UTF encoding in use, a universal character name sequence may result in a code point that consists of multiple characters, instead of a single normal `char` character.

When using the line feed escape sequence in text mode I/O, it is converted to the OS-specific newline byte or byte sequence.

The question mark escape sequence is used to avoid trigraphs. For example, `??/` is compiled as the trigraph representing a backslash character `'\'`, but using `?\?/` would result in the *string* `"??/"`.

There may be one, two or three octal numerals `n` in the octal value escape sequence.

Section 7.4: Integer literals

Integer literals are used to provide integral values. Three numerical bases are supported, indicated by prefixes:

Base	Prefix	Example
Decimal	None	`5`
Octal	`0`	`0345`
Hexadecimal	`0x` or `0X`	`0x12AB`, `0X12AB`, `0x12ab`, `0x12Ab`

Note that this writing doesn't include any sign, so integer literals are always positive. Something like `-1` is treated as an expression that has one integer literal (`1`) that is negated with a `-`

The type of a decimal integer literal is the first data type that can fit the value from `int` and `long`. Since C99, `long long` is also supported for very large literals.

The type of an octal or hexadecimal integer literal is the first data type that can fit the value from `int`, `unsigned`, `long`, and `unsigned long`. Since C99, `long long` and `unsigned long long` are also supported for very large literals.

Using various suffixes, the default type of a literal can be changed.

Suffix	Explanation
`L`, `l`	`long int`
`LL`, `ll` (since C99)	`long long int`
`U`, `u`	`unsigned`

The U and L/LL suffixes can be combined in any order and case. It is an error to duplicate suffixes (e.g. provide two `U` suffixes) even if they have different cases.

Chapter 8: Compound Literals

Section 8.1: Definition/Initialisation of Compound Literals

A compound literal is an unnamed object which is created in the scope where is defined. The concept was first introduced in C99 standard. An example for compound literal is

Examples from C standard, C11-§6.5.2.5/9:

> ```
> int *p = (int [2]){ 2, 4 };
> ```

p is initialized to the address of the first element of an unnamed array of two ints.

The compound literal is an lvalue. The storage duration of the unnamed object is either static (if the literal appears at file scope) or automatic (if the literal appears at block scope), and in the latter case the object's lifetime ends when control leaves the enclosing block.

> ```
> void f(void)
> {
>     int *p;
>     /*...*/
>     p = (int [2]){ *p };
>     /*...*/
> }
> ```
>
> p is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by p and the second, zero.[...]

Here p remains valid until the end of the block.

Compound literal with designators

(also from C11)

```
  struct point {
    unsigned x;
    unsigned y;
  };

  extern void drawline(struct point, struct point);

 // used somewhere like this
 drawline((struct point){.x=1, .y=1}, (struct point){.x=3, .y=4});
```

A fictive function `drawline` receives two arguments of type `struct point`. The first has coordinate values `x == 1` and `y == 1`, whereas the second has `x == 3` and `y == 4`

Compound literal without specifying array length

```
int *p = (int []){ 1, 2, 3};
```

In this case the size of the array is no specified then it will be determined by the length of the initializer.

Compound literal having length of initializer less than array size specified

```
int *p = (int [10]){1, 2, 3};
```

rest of the elements of compound literal will be initialized to `0` implicitly.

Read-only compound literal

Note that a compound literal is an lvalue and therefore it's elements can be modifiable. A *read-only* compound literal can be specified using `const` qualifier as `(const int[]){1,2}`.

Compound literal containing arbitrary expressions

Inside a function, a compound literal, as for any initialization since C99, can have arbitrary expressions.

```
void foo()
{
    int *p;
    int i = 2; j = 5;
    /*...*/
    p = (int [2]){ i+j, i*j };
    /*...*/
}
```

Chapter 9: Bit-fields

Parameter	Description
type-specifier	`signed`, `unsigned`, `int` or `_Bool`
identifier	The name for this field in the structure
size	The number of bits to use for this field

Most variables in C have a size that is an integral number of bytes. Bit-fields are a part of a structure that don't necessarily occupy a integral number of bytes; they can any number of bits. Multiple bit-fields can be packed into a single storage unit. They are a part of standard C, but there are many aspects that are implementation defined. They are one of the least portable parts of C.

Section 9.1: Bit-fields

A simple bit-field can be used to describe things that may have a specific number of bits involved.

```
struct encoderPosition {
   unsigned int encoderCounts : 23;
   unsigned int encoderTurns  : 4;
   unsigned int _reserved     : 5;
};
```

In this example we consider an encoder with 23 bits of single precision and 4 bits to describe multi-turn. Bit-fields are often used when interfacing with hardware that outputs data associated with specific number of bits. Another example could be communication with an FPGA, where the FPGA writes data into your memory in 32 bit sections allowing for hardware reads:

```
struct FPGAInfo {
    union {
        struct bits {
            unsigned int bulb1On  : 1;
            unsigned int bulb2On  : 1;
            unsigned int bulb1Off : 1;
            unsigned int bulb2Off : 1;
            unsigned int jetOn    : 1;
        };
        unsigned int data;
   };
};
```

For this example we have shown a commonly used construct to be able to access the data in its individual bits, or to write the data packet as a whole (emulating what the FPGA might do). We could then access the bits like this:

```
FPGAInfo fInfo;
fInfo.data = 0xFF34F;
if (fInfo.bits.bulb1On) {
    printf("Bulb 1 is on\n");
}
```

This is valid, but as per the C99 standard 6.7.2.1, item 10:

> The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined.

You need to be aware of endianness when defining bit-fields in this way. As such it may be necessary to use a preprocessor directive to check for the endianness of the machine. An example of this follows:

```
typedef union {
    struct bits {
#if defined(WIN32) || defined(LITTLE_ENDIAN)
    uint8_t commFailure :1;
    uint8_t hardwareFailure :1;
    uint8_t _reserved :6;
#else
    uint8_t _reserved :6;
    uint8_t hardwareFailure :1;
    uint8_t commFailure :1;
#endif
    };
    uint8_t data;
} hardwareStatus;
```

Section 9.2: Using bit-fields as small integers

```
#include <stdio.h>

int main(void)
{
    /* define a small bit-field that can hold values from 0 .. 7 */
    struct
    {
        unsigned int uint3: 3;
    } small;

    /* extract the right 3 bits from a value */
    unsigned int value = 255 - 2; /* Binary 11111101 */
    small.uint3 = value;          /* Binary      101 */
    printf("%d", small.uint3);

    /* This is in effect an infinite loop */
    for (small.uint3 = 0; small.uint3 < 8; small.uint3++)
    {
        printf("%d\n", small.uint3);
    }

    return 0;
}
```

Section 9.3: Bit-field alignment

Bit-fields give an ability to declare structure fields that are smaller than the character width. Bit-fields are implemented with byte-level or word-level mask. The following example results in a structure of 8 bytes.

```
struct C
{
    short s;            /* 2 bytes */
    char  c;            /* 1 byte */
    int   bit1 : 1;     /* 1 bit */
    int   nib  : 4;     /* 4 bits padded up to boundary of 8 bits. Thus 3 bits are padded */
    int   sept : 7;     /* 7 Bits septet, padded up to boundary of 32 bits. */
};
```

The comments describe one possible layout, but because the standard says *the alignment of the addressable storage unit is unspecified*, other layouts are also possible.

An unnamed bit-field may be of any size, but they can't be initialized or referenced.

A zero-width bit-field cannot be given a name and aligns the next field to the boundary defined by the datatype of the bit-field. This is achieved by padding bits between the bit-fields.

The size of structure 'A' is 1 byte.

```
struct A
{
    unsigned char c1 : 3;
    unsigned char c2 : 4;
    unsigned char c3 : 1;
};
```

In structure B, the first unnamed bit-field skips 2 bits; the zero width bit-field after `c2` causes `c3` to start from the char boundary (so 3 bits are skipped between `c2` and `c3`. There are 3 padding bits after `c4`. Thus the size of the structure is 2 bytes.

```
struct B
{
    unsigned char c1 : 1;
    unsigned char    : 2;    /* Skips 2 bits in the layout */
    unsigned char c2 : 2;
    unsigned char    : 0;    /* Causes padding up to next container boundary */
    unsigned char c3 : 4;
    unsigned char c4 : 1;
};
```

Section 9.4: Don'ts for bit-fields

1. Arrays of bit-fields, pointers to bit-fields and functions returning bit-fields are not allowed.
2. The address operator (&) cannot be applied to bit-field members.
3. The data type of a bit-field must be wide enough to contain the size of the field.
4. The `sizeof()` operator cannot be applied to a bit-field.
5. There is no way to create a `typedef` for a bit-field in isolation (though you can certainly create a `typedef` for a structure containing bit-fields).

```
typedef struct mybitfield
{
    unsigned char c1 : 20;   /* incorrect, see point 3 */
    unsigned char c2 : 4;    /* correct */
    unsigned char c3 : 1;
    unsigned int x[10]: 5;   /* incorrect, see point 1 */
} A;

int SomeFunction(void)
{
    // Somewhere in the code
    A a = { … };
    printf("Address of a.c2 is %p\n", &a.c2);      /* incorrect, see point 2 */
    printf("Size of a.c2 is %zu\n", sizeof(a.c2)); /* incorrect, see point 4 */
}
```

Section 9.5: When are bit-fields useful?

A bit-field is used to club together many variables into one object, similar to a structure. This allows for reduced memory usage and is especially useful in an embedded environment.

```
e.g. consider the following variables having the ranges as given below.
a --> range 0 - 3
b --> range 0 - 1
c --> range 0 - 7
d --> range 0 - 1
e --> range 0 - 1
```

If we declare these variables separately, then each has to be at least an 8-bit integer and the total space required will be 5 bytes. Moreover the variables will not use the entire range of an 8 bit unsigned integer (0-255). Here we can use bit-fields.

```
typedef struct {
   unsigned int a:2;
   unsigned int b:1;
   unsigned int c:3;
   unsigned int d:1;
   unsigned int e:1;
} bit_a;
```

The bit-fields in the structure are accessed the same as any other structure. The programmer needs to take care that the variables are written in range. If out of range the behaviour is undefined.

```
int main(void)
{
   bit_a bita_var;
   bita_var.a = 2;              // to write into element a
   printf ("%d",bita_var.a);    // to read from element a.
   return 0;
}
```

Often the programmer wants to zero the set of bit-fields. This can be done element by element, but there is second method. Simply create a union of the structure above with an unsigned type that is greater than, or equal to, the size of the structure. Then the entire set of bit-fields may be zeroed by zeroing this unsigned integer.

```
typedef union {
    struct {
       unsigned int a:2;
       unsigned int b:1;
       unsigned int c:3;
       unsigned int d:1;
       unsigned int e:1;
    };
    uint8_t data;
} union_bit;
```

Usage is as follows

```
int main(void)
{
   union_bit un_bit;
   un_bit.data = 0x00;        // clear the whole bit-field
   un_bit.a = 2;              // write into element a
```

```
    printf ("%d",un_bit.a);     // read from element a.
    return 0;
}
```

In conclusion, bit-fields are commonly used in memory constrained situations where you have a lot of variables which can take on limited ranges.

Chapter 10: Arrays

Arrays are derived data types, representing an ordered collection of values ("elements") of another type. Most arrays in C have a fixed number of elements of any one type, and its representation stores the elements contiguously in memory without gaps or padding. C allows multidimensional arrays whose elements are other arrays, and also arrays of pointers.

C supports dynamically allocated arrays whose size is determined at run time. C99 and later supports variable length arrays or VLAs.

Section 10.1: Declaring and initializing an array

The general syntax for declaring a one-dimensional array is

```
type arrName[size];
```

where `type` could be any built-in type or user-defined types such as structures, `arrName` is a user-defined identifier, and `size` is an integer constant.

Declaring an array (an array of 10 int variables in this case) is done like this:

```
int array[10];
```

it now holds indeterminate values. To ensure it holds zero values while declaring, you can do this:

```
int array[10] = {0};
```

Arrays can also have initializers, this example declares an array of 10 `int`'s, where the first 3 `int`'s will contain the values `1`, `2`, `3`, all other values will be zero:

```
int array[10] = {1, 2, 3};
```

In the above method of initialization, the first value in the list will be assigned to the first member of the array, the second value will be assigned to the second member of the array and so on. If the list size is smaller than the array size, then as in the above example, the remaining members of the array will be initialized to zeros. With designated list initialization (ISO C99), explicit initialization of the array members is possible. For example,

```
int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /* array is {0, 2, 5, 0, 9} */
```

In most cases, the compiler can deduce the length of the array for you, this can be achieved by leaving the square brackets empty:

```
int array[] = {1, 2, 3}; /* an array of 3 int's */
int array[] = {[3] = 8, [0] = 9}; /* size is 4 */
```

Declaring an array of zero length is not allowed.

`Version ≥ C99 Version < C11`

Variable Length Arrays (VLA for short) were added in C99, and made optional in C11. They are equal to normal arrays, with one, important, difference: The length doesn't have to be known at compile time. VLA's have automatic storage duration. Only pointers to VLA's can have static storage duration.

```c
size_t m = calc_length(); /* calculate array length at runtime */
int vla[m];               /* create array with calculated length */
```

Important:

VLA's are potentially dangerous. If the array `vla` in the example above requires more space on the stack than available, the stack will overflow. Usage of VLA's is therefore often discouraged in style guides and by books and exercises.

Section 10.2: Iterating through an array efficiently and row-major order

Arrays in C can be seen as a contiguous chunk of memory. More precisely, the last dimension of the array is the contiguous part. We call this the *row-major order*. Understanding this and the fact that a cache fault loads a complete cache line into the cache when accessing uncached data to prevent subsequent cache faults, we can see why accessing an array of dimension 10000x10000 with `array[0][0]` would **potentially** load `array[0][1]` in cache, but accessing `array[1][0]` right after would generate a second cache fault, since it is `sizeof(type)*10000` bytes away from `array[0][0]`, and therefore certainly not on the same cache line. Which is why iterating like this is inefficient:

```c
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[j][i] = 0;
    }
}
```

And iterating like this is more efficient:

```c
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[i][j] = 0;
    }
}
```

In the same vein, this is why when dealing with an array with one dimension and multiple indexes (let's say 2 dimensions here for simplicity with indexes i and j) it is important to iterate through the array like this:

```c
#define DIM_X 10
#define DIM_Y 20

int array[DIM_X*DIM_Y];

size_t i, j;
for (i = 0; i < DIM_X; ++i)
```

```
{
    for(j = 0; j < DIM_Y; ++j)
    {
        array[i*DIM_Y+j] = 0;
    }
}
```

Or with 3 dimensions and indexes i,j and k:

```
#define DIM_X 10
#define DIM_Y 20
#define DIM_Z 30

int array[DIM_X*DIM_Y*DIM_Z];

size_t i, j, k;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        for (k = 0; k < DIM_Z; ++k)
        {
            array[i*DIM_Y*DIM_Z+j*DIM_Z+k] = 0;
        }
    }
}
```

Or in a more generic way, when we have an array with **N1 x N2 x ... x Nd** elements, **d** dimensions and indices noted as **n1,n2,...,nd** the offset is calculated like this

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1)\cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

Picture/formula taken from: https://en.wikipedia.org/wiki/Row-major_order

Section 10.3: Array length

Arrays have fixed lengths that are known within the scope of their declarations. Nevertheless, it is possible and sometimes convenient to calculate array lengths. In particular, this can make code more flexible when the array length is determined automatically from an initializer:

```
int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

/* size of `array` in bytes */
size_t size = sizeof(array);

/* number of elements in `array` */
size_t length = sizeof(array) / sizeof(array[0]);
```

However, in most contexts where an array appears in an expression, it is automatically converted to ("decays to") a pointer to its first element. The case where an array is the operand of the `sizeof` operator is one of a small number of exceptions. The resulting pointer is not itself an array, and it does not carry any information about the length of the array from which it was derived. Therefore, if that length is needed in conjunction with the pointer, such as when the pointer is passed to a function, then it must be conveyed separately.

For example, suppose we want to write a function to return the last element of an array of `int`. Continuing from the

above, we might call it like so:

```
/* array will decay to a pointer, so the length must be passed separately */
int last = get_last(array, length);
```

The function could be implemented like this:

```
int get_last(int input[], size_t length) {
    return input[length - 1];
}
```

Note in particular that although the declaration of parameter `input` resembles that of an array, **it in fact declares `input` as a pointer** (to `int`). It is exactly equivalent to declaring `input` as `int *input`. The same would be true even if a dimension were given. This is possible because arrays cannot ever be actual arguments to functions (they decay to pointers when they appear in function call expressions), and it can be viewed as mnemonic.

It is a very common error to attempt to determine array size from a pointer, which cannot work. DO NOT DO THIS:

```
int BAD_get_last(int input[]) {
    /* INCORRECTLY COMPUTES THE LENGTH OF THE ARRAY INTO WHICH input POINTS: */
    size_t length = sizeof(input) / sizeof(input[0]));

    return input[length - 1];  /* Oops -- not the droid we are looking for */
}
```

In fact, that particular error is so common that some compilers recognize it and warn about it. `clang`, for instance, will emit the following warning:

```
warning: sizeof on array function parameter will return size of 'int *' instead of 'int []' [-
Wsizeof-array-argument]
        int length = sizeof(input) / sizeof(input[0]);
                           ^
note: declared here
int BAD_get_last(int input[])
                     ^
```

Section 10.4: Passing multidimensional arrays to a function

Multidimensional arrays follow the same rules as single-dimensional arrays when passing them to a function. However the combination of decay-to-pointer, operator precedence, and the two different ways to declare a multidimensional array (array of arrays vs array of pointers) may make the declaration of such functions non-intuitive. The following example shows the correct ways to pass multidimensional arrays.

```
#include <assert.h>
#include <stdlib.h>

/* When passing a multidimensional array (i.e. an array of arrays) to a
   function, it decays into a pointer to the first element as usual.  But only
   the top level decays, so what is passed is a pointer to an array of some fixed
   size (4 in this case). */
void f(int x[][4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* This prototype is equivalent to f(int x[][4]).
   The parentheses around *x are required because [index] has a higher
   precedence than *expr, thus int *x[4] would normally be equivalent to int
```

```
   *(x[4]), i.e. an array of 4 pointers to int.  But if it's declared as a
   function parameter, it decays into a pointer and becomes int **x,
   which is not compatable with x[2][4]. */
void g(int (*x)[4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* An array of pointers may be passed to this, since it'll decay into a pointer
   to pointer, but an array of arrays may not. */
void h(int **x) {
    assert(sizeof(*x) == sizeof(int*));
}

int main(void) {
    int foo[2][4];
    f(foo);
    g(foo);

    /* Here we're dynamically creating an array of pointers.  Note that the
       size of each dimension is not part of the datatype, and so the type
       system just treats it as a pointer to pointer, not a pointer to array
       or array of arrays. */
    int **bar = malloc(sizeof(*bar) * 2);
    assert(bar);
    for (size_t i = 0; i < 2; i++) {
        bar[i] = malloc(sizeof(*bar[i]) * 4);
        assert(bar[i]);
    }

    h(bar);

    for (size_t i = 0; i < 2; i++) {
        free(bar[i]);
    }
    free(bar);
}
```

See also

Passing in Arrays to Functions

Section 10.5: Multi-dimensional arrays

The C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration −

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional (5 x 10 x 4) integer array:

```
int arr[5][10][4];
```

Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of dimensions m x n, we can write as follows:

```
type arrayName[m][n];
```

Where `type` can be any valid C data type (`int`, `float`, etc.) and `arrayName` can be any valid C identifier. A two-dimensional array can be visualized as a table with `m` rows and `n` columns. **Note**: The order *does* matter in C. The array `int a[4][3]` is not the same as the array `int a[3][4]`. The number of rows comes first as C is a *row*-major language.

A two-dimensional array `a`, which contains three rows and four columns can be shown as follows:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array `a` is identified by an element name of the form `a[i][j]`, where `a` is the name of the array, `i` represents which row, and `j` represents which column. Recall that rows and columns are zero indexed. This is very similar to mathematical notation for subscripting 2-D matrices.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. The following define an array with 3 rows where each row has 4 columns.

```
int a[3][4] = {
   {0, 1, 2, 3} ,    /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,    /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}    /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

While the method of creating arrays with nested braces is optional, it is strongly encouraged as it is more readable and clearer.

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. Let us check the following program where we have used a nested loop to handle a two-dimensional array:

```
#include <stdio.h>

int main () {

   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
```

```
    int i, j;

    /* output each array element's value */
    for ( i = 0; i < 5; i++ ) {

        for ( j = 0; j < 2; j++ ) {
            printf("a[%d][%d] = %d\n", i,j, a[i][j] );
        }
    }

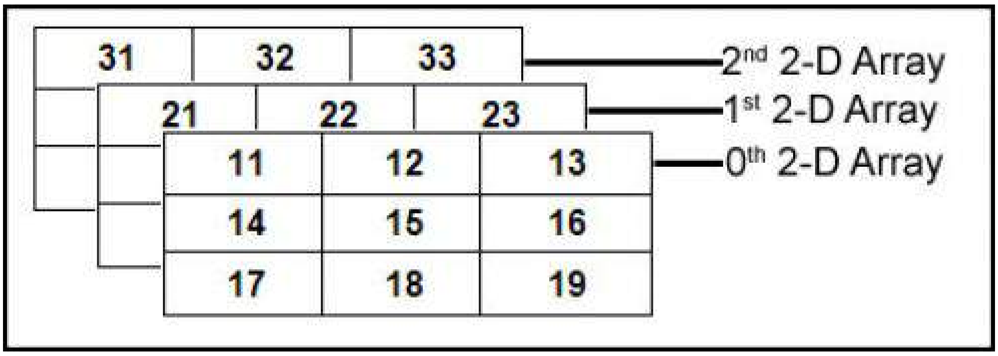
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

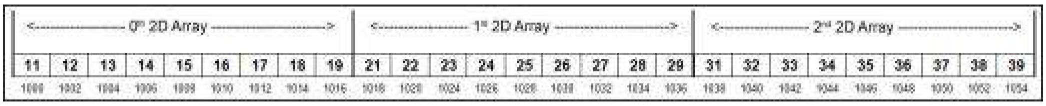
```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Three-Dimensional array:

A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D arrays.



3D array memory map:



Initializing a 3D Array:

```
double cprogram[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
 {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
 {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};
```

We can have arrays with any number of dimensions, although it is likely that most of the arrays that are created will be of one or two dimensions.

Section 10.6: Define array and access array element

```
#include <stdio.h>

#define ARRLEN (10)

int main (void)
{

   int n[ ARRLEN ]; /* n is an array of 10 integers */
   size_t i, j; /* Use size_t to address memory, that is to index arrays, as its guaranteed to
                   be wide enough to address all of the possible available memory.
                   Using signed integers to do so should be considered a special use case,
                   and should be restricted to the uncommon case of being in the need of
                   negative indexes. */

   /* Initialize elements of array n. */
   for ( i = 0; i < ARRLEN ; i++ )
   {
      n[ i ] = i + 100; /* Set element at location i to i + 100. */
   }

   /* Output each array element's value. */
   for (j = 0; j < ARRLEN ; j++ )
   {
      printf("Element[%zu] = %d\n", j, n[j] );
   }

   return 0;
}
```

Section 10.7: Clearing array contents (zeroing)

Sometimes it's necessary to set an array to zero, after the initialization has been done.

```
#include <stdlib.h> /* for EXIT_SUCCESS */

#define ARRLEN (10)

int main(void)
{
  int array[ARRLEN]; /* Allocated but not initialised, as not defined static or global. */

  size_t i;
  for(i = 0; i < ARRLEN; ++i)
  {
    array[i] = 0;
  }

  return EXIT_SUCCESS;
}
```

An common short cut to the above loop is to use `memset()` from **`<string.h>`**. Passing `array` as shown below makes it decay to a pointer to its 1st element.

```
memset(array, 0, ARRLEN * sizeof (int)); /* Use size explicitly provided type (int here). */
```

or

```c
memset(array, 0, ARRLEN * sizeof *array); /* Use size of type the pointer is pointing to. */
```

As in this example `array` *is* an array and not just a pointer to an array's 1st element (see Array length on why this is important) a third option to 0-out the array is possible:

```c
 memset(array, 0, sizeof array); /* Use size of the array itself. */
```

Section 10.8: Setting values in arrays

Accessing array values is generally done through square brackets:

```c
int val;
int array[10];

/* Setting the value of the fifth element to 5: */
array[4] = 5;

/* The above is equal to: */
*(array + 4) = 5;

/* Reading the value of the fifth element: */
val = array[4];
```

As a side effect of the operands to the `+` operator being exchangeable (--> commutative law) the following is equivalent:

```c
*(array + 4) = 5;
*(4 + array) = 5;
```

so as well the next statements are equivalent:

```c
array[4] = 5;
4[array] = 5; /* Weird but valid C ... */
```

and those two as well:

```c
val = array[4];
val = 4[array]; /* Weird but valid C ... */
```

C doesn't perform any boundary checks, accessing contents outside of the declared array is undefined (Accessing memory beyond allocated chunk):

```c
int val;
int array[10];

array[4] = 5;    /* ok */
val = array[4];  /* ok */
array[19] = 20;  /* undefined behavior */
val = array[15]; /* undefined behavior */
```

Section 10.9: Allocate and zero-initialize an array with user defined size

```c
#include <stdio.h>
#include <stdlib.h>
```

```c
int main (void)
{
  int * pdata;
  size_t n;

  printf ("Enter the size of the array: ");
  fflush(stdout); /* Make sure the prompt gets printed to buffered stdout. */

  if (1 != scanf("%zu", &n)) /* If zu is not supported (Windows?) use lu. */
  {
    fprintf("scanf() did not read a in proper value.\n");
    exit(EXIT_FAILURE);
  }

  pdata = calloc(n, sizeof *pdata);
  if (NULL == pdata)
  {
    perror("calloc() failed"); /* Print error. */
    exit(EXIT_FAILURE);
  }

  free(pdata); /* Clean up. */

  return EXIT_SUCCESS;
}
```

This program tries to scan in an unsigned integer value from standard input, allocate a block of memory for an array of n elements of type `int` by calling the `calloc()` function. The memory is initialized to all zeros by the latter.

In case of success the memory is releases by the call to `free()`.

Section 10.10: Iterating through an array using pointers

```c
#include <stdio.h>
#define SIZE (10)
int main()
{
    size_t i = 0;
    int *p = NULL;
    int a[SIZE];

    /* Setting up the values to be i*i */
    for(i = 0; i < SIZE; ++i)
    {
        a[i] = i * i;
    }

    /* Reading the values using pointers */
    for(p = a; p < a + SIZE; ++p)
    {
        printf("%d\n", *p);
    }

    return 0;
}
```

Here, in the initialization of p in the first `for` loop condition, the array a *decays* to a pointer to its first element, as it would in almost all places where such an array variable is used.

Then, the `++p` performs pointer arithmetic on the pointer p and walks one by one through the elements of the

array, and refers to them by dereferencing them with `*p`.

Chapter 11: Linked lists

Section 11.1: A doubly linked list

An example of code showing how nodes can be inserted at a doubly linked list, how the list can easily be reversed, and how it can be printed in reverse.

```
#include <stdio.h>
#include <stdlib.h>

/* This data is not always stored in a structure, but it is sometimes for ease of use */
struct Node {
  /* Sometimes a key is also stored and used in the functions */
  int data;
  struct Node* next;
  struct Node* previous;
};

void insert_at_beginning(struct Node **pheadNode, int value);
void insert_at_end(struct Node **pheadNode, int value);

void print_list(struct Node *headNode);
void print_list_backwards(struct Node *headNode);

void free_list(struct Node *headNode);

int main(void) {
  /* Sometimes in a doubly linked list the last node is also stored */
  struct Node *head = NULL;

  printf("Insert a node at the beginning of the list.\n");
  insert_at_beginning(&head, 5);
  print_list(head);

  printf("Insert a node at the beginning, and then print the list backwards\n");
  insert_at_beginning(&head, 10);
  print_list_backwards(head);

  printf("Insert a node at the end, and then print the list forwards.\n");

  insert_at_end(&head, 15);
  print_list(head);

  free_list(head);

  return 0;
}

void print_list_backwards(struct Node *headNode) {
  if (NULL == headNode)
  {
    return;
  }
  /*
  Iterate through the list, and once we get to the end, iterate backwards to print
  out the items in reverse order (this is done with the pointer to the previous node).
  This can be done even more easily if a pointer to the last node is stored.
  */
  struct Node *i = headNode;
  while (i->next != NULL) {
```

```
    i = i->next; /* Move to the end of the list */
  }

  while (i != NULL) {
    printf("Value: %d\n", i->data);
    i = i->previous;
  }
}

void print_list(struct Node *headNode) {
  /* Iterate through the list and print out the data member of each node */
  struct Node *i;
  for (i = headNode; i != NULL; i = i->next) {
    printf("Value: %d\n", i->data);
  }
}

void insert_at_beginning(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }
  /*
  This is done similarly to how we insert a node at the beginning of a singly linked
  list, instead we set the previous member of the structure as well
  */
  currentNode = malloc(sizeof *currentNode);

  currentNode->next = NULL;
  currentNode->previous = NULL;
  currentNode->data = value;

  if (*pheadNode == NULL) { /* The list is empty */
    *pheadNode = currentNode;
    return;
  }

  currentNode->next = *pheadNode;
  (*pheadNode)->previous = currentNode;
  *pheadNode = currentNode;
}

void insert_at_end(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }

  /*
  This can, again be done easily by being able to have the previous element.  It
  would also be even more useful to have a pointer to the last node, which is commonly
  used.
  */

  currentNode = malloc(sizeof *currentNode);
  struct Node *i = *pheadNode;

  currentNode->data = value;
```

```c
  currentNode->next = NULL;
  currentNode->previous = NULL;

  if (*pheadNode == NULL) {
    *pheadNode = currentNode;
    return;
  }

  while (i->next != NULL) { /* Go to the end of the list */
    i = i->next;
  }

  i->next = currentNode;
  currentNode->previous = i;
}

void free_list(struct Node *node) {
  while (node != NULL) {
    struct Node *next = node->next;
    free(node);
    node = next;
  }
}
```

Note that sometimes, storing a pointer to the last node is useful (it is more efficient to simply be able to jump straight to the end of the list than to need to iterate through to the end):

```c
struct Node *lastNode = NULL;
```

In which case, updating it upon changes to the list is needed.

Sometimes, a key is also used to identify elements. It is simply a member of the Node structure:

```c
struct Node {
  int data;
  int key;
  struct Node* next;
  struct Node* previous;
};
```

The key is then used when any tasks are performed on a specific element, like deleting elements.

Section 11.2: Reversing a linked list

You can also perform this task recursively, but I have chosen in this example to use an iterative approach. This task is useful if you are inserting all of your nodes at the beginning of a linked list. Here is an example:

```c
#include <stdio.h>
#include <stdlib.h>

#define NUM_ITEMS 10

struct Node {
  int data;
  struct Node *next;
};

void insert_node(struct Node **headNode, int nodeValue, int position);
void print_list(struct Node *headNode);
```

```c
void reverse_list(struct Node **headNode);


int main(void) {
  int i;
  struct Node *head = NULL;

  for(i = 1; i <= NUM_ITEMS; i++) {
    insert_node(&head, i, i);
  }
  print_list(head);

  printf("I will now reverse the linked list\n");
  reverse_list(&head);
  print_list(head);
  return 0;
}

void print_list(struct Node *headNode) {
  struct Node *iterator;

  for(iterator = headNode; iterator != NULL; iterator = iterator->next) {
    printf("Value: %d\n", iterator->data);
  }
}

void insert_node(struct Node **headNode, int nodeValue, int position) {
  int i;
  struct Node *currentNode = (struct Node *)malloc(sizeof(struct Node));
  struct Node *nodeBeforePosition = *headNode;

  currentNode->data = nodeValue;

  if(position == 1) {
    currentNode->next = *headNode;
    *headNode = currentNode;
    return;
  }

  for (i = 0; i < position - 2; i++) {
    nodeBeforePosition = nodeBeforePosition->next;
  }

  currentNode->next = nodeBeforePosition->next;
  nodeBeforePosition->next = currentNode;
}

void reverse_list(struct Node **headNode) {
  struct Node *iterator = *headNode;
  struct Node *previousNode = NULL;
  struct Node *nextNode = NULL;

  while (iterator != NULL) {
    nextNode = iterator->next;
    iterator->next = previousNode;
    previousNode = iterator;
    iterator = nextNode;
  }

  /* Iterator will be NULL by the end, so the last node will be stored in
  previousNode.  We will set the last node to be the headNode */
  *headNode = previousNode;
```

```
}
```

Explanation for the Reverse List Method

We start the `previousNode` out as `NULL`, since we know on the first iteration of the loop, if we are looking for the node before the first head node, it will be `NULL`. The first node will become the last node in the list, and the next variable should naturally be `NULL`.

Basically, the concept of reversing the linked list here is that we actually reverse the links themselves. Each node's next member will become the node before it, like so:

```
Head -> 1 -> 2 -> 3 -> 4 -> 5
```

Where each number represents a node. This list would become:

```
1 <- 2 <- 3 <- 4 <- 5 <- Head
```

Finally, the head should point to the 5th node instead, and each node should point to the node previous of it.

Node 1 should point to `NULL` since there was nothing before it. Node 2 should point to node 1, node 3 should point to node 2, et cetera.

However, there is *one small problem* with this method. If we break the link to the next node and change it to the previous node, we will not be able to traverse to the next node in the list since the link to it is gone.

The solution to this problem is to simply store the next element in a variable (`nextNode`) before changing the link.

Section 11.3: Inserting a node at the nth position

So far, we have looked at inserting a node at the beginning of a singly linked list. However, most of the times you will want to be able to insert nodes elsewhere as well. The code written below shows how it is possible to write an `insert()` function to insert nodes *anywhere* in the linked lists.

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};

struct Node* insert(struct Node* head, int value, size_t position);
void print_list (struct Node* head);

int main(int argc, char *argv[]) {
  struct Node *head = NULL; /* Initialize the list to be empty */

  /* Insert nodes at positions with values: */
  head = insert(head, 1, 0);
  head = insert(head, 100, 1);
  head = insert(head, 21, 2);
  head = insert(head, 2, 3);
  head = insert(head, 5, 4);
  head = insert(head, 42, 2);

  print_list(head);
  return 0;
```

```
}

struct Node* insert(struct Node* head, int value, size_t position) {
  size_t i = 0;
  struct Node *currentNode;

  /* Create our node */
  currentNode = malloc(sizeof *currentNode);
  /* Check for success of malloc() here! */

  /* Assign data */
  currentNode->data = value;

  /* Holds a pointer to the 'next' field that we have to link to the new node.
     By initializing it to &head we handle the case of insertion at the beginning. */
  struct Node **nextForPosition = &head;
  /* Iterate to get the 'next' field we are looking for.
     Note: Insert at the end if position is larger than current number of elements. */
  for (i = 0; i < position && *nextForPosition != NULL; i++) {
      /* nextForPosition is pointing to the 'next' field of the node.
         So *nextForPosition is a pointer to the next node.
         Update it with a pointer to the 'next' field of the next node. */
      nextForPosition = &(*nextForPosition)->next;
  }

  /* Here, we are taking the link to the next node (the one our newly inserted node should
  point to) by dereferencing nextForPosition, which points to the 'next' field of the node
  that is in the position we want to insert our node at.
  We assign this link to our next value. */
  currentNode->next = *nextForPosition;

  /* Now, we want to correct the link of the node before the position of our
  new node: it will be changed to be a pointer to our new node. */
  *nextForPosition = currentNode;

  return head;
}

void print_list (struct Node* head) {
  /* Go through the list of nodes and print out the data in each node */
  struct Node* i = head;
  while (i != NULL) {
    printf("%d\n", i->data);
    i = i->next;
  }
}
```

Section 11.4: Inserting a node at the beginning of a singly linked list

The code below will prompt for numbers and continue to add them to the beginning of a linked list.

```
/* This program will demonstrate inserting a node at the beginning of a linked list */

#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};
```

```c
void insert_node (struct Node **head, int nodeValue);
void print_list (struct Node *head);

int main(int argc, char *argv[]) {
  struct Node* headNode;
  headNode = NULL; /* Initialize our first node pointer to be NULL. */
  size_t listSize, i;
  do {
    printf("How many numbers would you like to input?\n");
  } while(1 != scanf("%zu", &listSize));

  for (i = 0; i < listSize; i++) {
    int numToAdd;
    do {
      printf("Enter a number:\n");
    } while (1 != scanf("%d", &numToAdd));

    insert_node (&headNode, numToAdd);
    printf("Current list after your inserted node: \n");
    print_list(headNode);
  }

  return 0;
}

void print_list (struct Node *head) {
  struct node* currentNode = head;

  /* Iterate through each link. */
  while (currentNode != NULL) {
      printf("Value: %d\n", currentNode->data);
      currentNode = currentNode -> next;
  }
}

void insert_node (struct Node **head, int nodeValue) {
  struct Node *currentNode = malloc(sizeof *currentNode);
  currentNode->data = nodeValue;
  currentNode->next = (*head);

  *head = currentNode;
}
```

Explanation for the Insertion of Nodes

In order to understand how we add nodes at the beginning, let's take a look at possible scenarios:

1. The list is empty, so we need to add a new node. In which case, our memory looks like this where `HEAD` is a pointer to the first node:

```
| HEAD | --> NULL
```

The line `currentNode->next = *headNode;` will assign the value of `currentNode->next` to be `NULL` since `headNode` originally starts out at a value of `NULL`.

Now, we want to set our head node pointer to point to our current node.

```
  -----      -------------
|HEAD | --> |CURRENTNODE| --> NULL /* The head node points to the current node */
```

```
-----      -------------
```

This is done with `*headNode = currentNode;`

2. The list is already populated; we need to add a new node to the beginning. For the sake of simplicity, let's start out with 1 node:

```
 -----    -----------
HEAD --> FIRST NODE --> NULL
-----    -----------
```

With `currentNode->next = *headNode`, the data structure looks like this:

```
 ---------        -----    ---------------------
currentNode -->  HEAD --> POINTER TO FIRST NODE --> NULL
---------        -----    ---------------------
```

Which, obviously needs to be altered since `*headNode` should point to `currentNode`.

```
 ----    -----------    ---------------
HEAD -> currentNode -->     NODE       -> NULL
----    -----------    ---------------
```

This is done with `*headNode = currentNode;`

Chapter 12: Enumerations

Section 12.1: Simple Enumeration

An enumeration is a user-defined data type consists of integral constants and each integral constant is given a name. Keyword **enum** is used to define enumerated data type.

If you use **enum** instead of `int` or `string/ char*`, you increase compile-time checking and avoid errors from passing in invalid constants, and you document which values are legal to use.

Example 1

```
enum color{ RED, GREEN, BLUE };

void printColor(enum color chosenColor)
{
    const char *color_name = "Invalid color";
    switch (chosenColor)
    {
       case RED:
         color_name = "RED";
         break;

       case GREEN:
        color_name = "GREEN";
        break;

       case BLUE:
        color_name = "BLUE";
        break;
    }
    printf("%s\n", color_name);
}
```

With a main function defined as follows (for example):

```
int main(){
    enum color chosenColor;
    printf("Enter a number between 0 and 2");
    scanf("%d", (int*)&chosenColor);
    printColor(chosenColor);
    return 0;
}
```

Version ≥ C99

Example 2

(This example uses designated initializers which are standardized since C99.)

```
enum week{ MON, TUE, WED, THU, FRI, SAT, SUN };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
   printf("%s\n", dow[day]);
}
```

The same example using range checking:

```
enum week{ DOW_INVALID = -1,
  MON, TUE, WED, THU, FRI, SAT, SUN,
  DOW_MAX };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
   assert(day > DOW_INVALID && day < DOW_MAX);
   printf("%s\n", dow[day]);
}
```

Section 12.2: enumeration constant without typename

Enumeration types can also be declared without giving them a name:

```
  enum { buffersize = 256, };
  static unsigned char buffer [buffersize] = { 0 };
```

This enables us to define compile time constants of type `int` that can as in this example be used as array length.

Section 12.3: Enumeration with duplicate value

An enumerations value in no way needs to be unique:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */


enum Dupes
{
   Base, /* Takes 0 */
   One, /* Takes Base + 1 */
   Two, /* Takes One + 1 */
   Negative = -1,
   AnotherZero /* Takes Negative + 1 == 0, sigh */
};

int main(void)
{
  printf("Base = %d\n", Base);
  printf("One = %d\n", One);
  printf("Two = %d\n", Two);
  printf("Negative = %d\n", Negative);
  printf("AnotherZero = %d\n", AnotherZero);

  return EXIT_SUCCESS;
}
```

The sample prints:

```
Base = 0
One = 1
Two = 2
```

```
Negative = -1
AnotherZero = 0
```

Section 12.4: Typedef enum

There are several possibilities and conventions to name an enumeration. The first is to use a *tag name* just after the **enum** keyword.

```
enum color
{
    RED,
    GREEN,
    BLUE
};
```

This enumeration must then always be used with the keyword *and* the tag like this:

```
enum color chosenColor = RED;
```

If we use `typedef` directly when declaring the **enum**, we can omit the tag name and then use the type without the **enum** keyword:

```
typedef enum
{
    RED,
    GREEN,
    BLUE
} color;

color chosenColor = RED;
```

But in this latter case we cannot use it as `enum color`, because we didn't use the tag name in the definition. One common convention is to use both, such that the same name can be used with or without **enum** keyword. This has the particular advantage of being compatible with C++

```
enum color                /* as in the first example */
{
    RED,
    GREEN,
    BLUE
};
typedef enum color color; /* also a typedef of same identifier */

color chosenColor  = RED;
enum color defaultColor = BLUE;
```

Function:

```
void printColor()
{
    if (chosenColor == RED)
    {
        printf("RED\n");
    }
    else if (chosenColor == GREEN)
    {
        printf("GREEN\n");
```

```
    }
    else if (chosenColor == BLUE)
    {
        printf("BLUE\n");
    }
}
```

For more on `typedef` see Typedef

Chapter 13: Structs

Structures provide a way to group a set of related variables of diverse types into a single unit of memory. The structure as a whole can be referenced by a single name or pointer; the structure members can be accessed individually too. Structures can be passed to functions and returned from functions. They are defined using the keyword `struct`.

Section 13.1: Flexible Array Members

Version ≥ C99

Type Declaration

A structure *with at least one member* may additionally contain a single array member of unspecified length at the end of the structure. This is called a flexible array member:

```
struct ex1
{
    size_t foo;
    int flex[];
};

struct ex2_header
{
    int foo;
    char bar;
};

struct ex2
{
    struct ex2_header hdr;
    int flex[];
};

/* Merged ex2_header and ex2 structures. */
struct ex3
{
    int foo;
    char bar;
    int flex[];
};
```

Effects on Size and Padding

A flexible array member is treated as having no size when calculating the size of a structure, though padding between that member and the previous member of the structure may still exist:

```
/* Prints "8,8" on my machine, so there is no padding. */
printf("%zu,%zu\n", sizeof(size_t), sizeof(struct ex1));

/* Also prints "8,8" on my machine, so there is no padding in the ex2 structure itself. */
printf("%zu,%zu\n", sizeof(struct ex2_header), sizeof(struct ex2));

/* Prints "5,8" on my machine, so there are 3 bytes of padding. */
printf("%zu,%zu\n", sizeof(int) + sizeof(char), sizeof(struct ex3));
```

The flexible array member is considered to have an incomplete array type, so its size cannot be calculated using `sizeof`.

Usage

You can declare and initialize an object with a structure type containing a flexible array member, but you must not attempt to initialize the flexible array member since it is treated as if it does not exist. It is forbidden to try to do this, and compile errors will result.

Similarly, you should not attempt to assign a value to any element of a flexible array member when declaring a structure in this way since there may not be enough padding at the end of the structure to allow for any objects required by the flexible array member. The compiler will not necessarily prevent you from doing this, however, so this can lead to undefined behavior.

```
/* invalid: cannot initialize flexible array member */
struct ex1 e1 = {1, {2, 3}};
/* invalid: hdr={foo=1, bar=2} OK, but cannot initialize flexible array member */
struct ex2 e2 = {{1, 2}, {3}};
/* valid: initialize foo=1, bar=2 members */
struct ex3 e3 = {1, 2};

e1.flex[0] = 3; /* undefined behavior, in my case */
e3.flex[0] = 2; /* undefined behavior again */
e2.flex[0] = e3.flex[0]; /* undefined behavior */
```

You may instead choose to use `malloc`, `calloc`, or `realloc` to allocate the structure with extra storage and later free it, which allows you to use the flexible array member as you wish:

```
/* valid: allocate an object of structure type `ex1` along with an array of 2 ints */
struct ex1 *pe1 = malloc(sizeof(*pe1) + 2 * sizeof(pe1->flex[0]));

/* valid: allocate an object of structure type ex2 along with an array of 4 ints */
struct ex2 *pe2 = malloc(sizeof(struct ex2) + sizeof(int[4]));

/* valid: allocate 5 structure type ex3 objects along with an array of 3 ints per object */
struct ex3 *pe3 = malloc(5 * (sizeof(*pe3) + sizeof(int[3])));

pe1->flex[0] = 3; /* valid */
pe3[0]->flex[0] = pe1->flex[0]; /* valid */
```

Version < C99

The 'struct hack'

Flexible array members did not exist prior to C99 and are treated as errors. A common workaround is to declare an array of length 1, a technique called the 'struct hack':

```
struct ex1
{
    size_t foo;
    int flex[1];
};
```

This will affect the size of the structure, however, unlike a true flexible array member:

```
/* Prints "8,4,16" on my machine, signifying that there are 4 bytes of padding. */
printf("%d,%d,%d\n", (int)sizeof(size_t), (int)sizeof(int[1]), (int)sizeof(struct ex1));
```

To use the `flex` member as a flexible array member, you'd allocate it with `malloc` as shown above, except that `sizeof(*pe1)` (or the equivalent `sizeof(struct ex1)`) would be replaced with `offsetof(struct ex1, flex)` or the longer, type-agnostic expression `sizeof(*pe1)-sizeof(pe1->flex)`. Alternatively, you might subtract 1 from the desired length of the "flexible" array since it's already included in the structure size, assuming the desired length is

greater than 0. The same logic may be applied to the other usage examples.

Compatibility

If compatibility with compilers that do not support flexible array members is desired, you may use a macro defined like `FLEXMEMB_SIZE` below:

```
#if __STDC_VERSION__ < 199901L
#define FLEXMEMB_SIZE 1
#else
#define FLEXMEMB_SIZE /* nothing */
#endif

struct ex1
{
    size_t foo;
    int flex[FLEXMEMB_SIZE];
};
```

When allocating objects, you should use the `offsetof(struct ex1, flex)` form to refer to the structure size (excluding the flexible array member) since it is the only expression that will remain consistent between compilers that support flexible array members and compilers that do not:

```
struct ex1 *pe10 = malloc(offsetof(struct ex1, flex) + n * sizeof(pe10->flex[0]));
```

The alternative is to use the preprocessor to conditionally subtract 1 from the specified length. Due to the increased potential for inconsistency and general human error in this form, I moved the logic into a separate function:

```
struct ex1 *ex1_alloc(size_t n)
{
    struct ex1 tmp;
#if __STDC_VERSION__ < 199901L
    if (n != 0)
        n--;
#endif
    return malloc(sizeof(tmp) + n * sizeof(tmp.flex[0]));
}
...

/* allocate an ex1 object with "flex" array of length 3 */
struct ex1 *pe1 = ex1_alloc(3);
```

Section 13.2: Typedef Structs

Combining `typedef` with `struct` can make code clearer. For example:

```
typedef struct
{
    int x, y;
} Point;
```

as opposed to:

```
struct Point
{
    int x, y;
};
```

could be declared as:

```
Point point;
```

instead of:

```
struct Point point;
```

Even better is to use the following

```
typedef struct Point Point;

struct Point
{
    int x, y;
};
```

to have advantage of both possible definitions of `point`. Such a declaration is most convenient if you learned C++ first, where you may omit the `struct` keyword if the name is not ambiguous.

`typedef` names for structs could be in conflict with other identifiers of other parts of the program. Some consider this a disadvantage, but for most people having a `struct` and another identifier the same is quite disturbing. Notorious is e.g POSIX' `stat`

```
int stat(const char *pathname, struct stat *buf);
```

where you see a function `stat` that has one argument that is `struct stat`.

`typedef`'d structs without a tag name always impose that the whole `struct` declaration is visible to code that uses it. The entire `struct` declaration must then be placed in a header file.

Consider:

```
#include "bar.h"

struct foo
{
    bar *aBar;
};
```

So with a `typedef`d `struct` that has no tag name, the `bar.h` file always has to include the whole definition of `bar`. If we use

```
typedef struct bar bar;
```

in `bar.h`, the details of the `bar` structure can be hidden.

See Typedef

Section 13.3: Pointers to structs

When you have a variable containing a `struct`, you can access its fields using the dot operator (`.`). However, if you have a pointer to a `struct`, this will not work. You have to use the arrow operator (`->`) to access its fields. Here's an example of a terribly simple (some might say "terrible and simple") implementation of a stack that uses pointers to `struct`s and demonstrates the arrow operator.

```
#include <stdlib.h>
#include <stdio.h>

/* structs */
struct stack
{
    struct node *top;
    int size;
};

struct node
{
    int data;
    struct node *next;
};

/* function declarations */
int push(int, struct stack*);
int pop(struct stack*);
void destroy(struct stack*);

int main(void)
{
    int result = EXIT_SUCCESS;

    size_t i;

    /* allocate memory for a struct stack and record its pointer */
    struct stack *stack = malloc(sizeof *stack);
    if (NULL == stack)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    /* initialize stack */
    stack->top = NULL;
    stack->size = 0;

    /* push 10 ints */
    {
        int data = 0;
        for(i = 0; i < 10; i++)
        {
            printf("Pushing: %d\n", data);
            if (-1 == push(data, stack))
            {
                perror("push() failed");
                result = EXIT_FAILURE;
                break;
            }

            ++data;
        }
    }

    if (EXIT_SUCCESS == result)
    {
        /* pop 5 ints */
        for(i = 0; i < 5; i++)
        {
            printf("Popped: %i\n", pop(stack));
```

```c
        }
    }

    /* destroy stack */
    destroy(stack);

    return result;
}

/* Push a value onto the stack. */
/* Returns 0 on success and -1 on failure. */
int push(int data, struct stack *stack)
{
    int result = 0;

    /* allocate memory for new node */
    struct node *new_node = malloc(sizeof *new_node);
    if (NULL == new_node)
    {
        result = -1;
    }
    else
    {
        new_node->data = data;
        new_node->next = stack->top;
        stack->top = new_node;
        stack->size++;
    }

    return result;
}

/* Pop a value off of the stack. */
/* Returns the value popped off the stack */
int pop(struct stack *stack)
{
    struct node *top = stack->top;
    int data = top->data;
    stack->top = top->next;
    stack->size--;
    free(top);
    return data;
}

/* destroy the stack */
void destroy(struct stack *stack)
{
    /* free all pointers */
    while(stack->top != NULL)
    {
        pop(stack);
    }
}
```

Section 13.4: Passing structs to functions

In C, all arguments are passed to functions by value, including structs. For small structs, this is a good thing as it means there is no overhead from accessing the data through a pointer. However, it also makes it very easy to accidentally pass a huge struct resulting in poor performance, particularly if the programmer is used to other languages where arguments are passed by reference.

```c
struct coordinates
{
    int x;
    int y;
    int z;
};

// Passing and returning a small struct by value, very fast
struct coordinates move(struct coordinates position, struct coordinates movement)
{
    position.x += movement.x;
    position.y += movement.y;
    position.z += movement.z;
    return position;
}

// A very big struct
struct lotsOfData
{
    int param1;
    char param2[80000];
};

// Passing and returning a large struct by value, very slow!
// Given the large size of the struct this could even cause stack overflow
struct lotsOfData doubleParam1(struct lotsOfData value)
{
    value.param1 *= 2;
    return value;
}

// Passing the large struct by pointer instead, fairly fast
void doubleParam1ByPtr(struct lotsOfData *value)
{
    value->param1 *= 2;
}
```

Section 13.5: Object-based programming using structs

Structs may be used to implement code in an object oriented manner. A struct is similar to a class, but is missing the functions which normally also form part of a class, we can add these as function pointer member variables. To stay with our coordinates example:

```c
/* coordinates.h */

typedef struct coordinate_s
{
    /* Pointers to method functions */
    void (*setx)(coordinate *this, int x);
    void (*sety)(coordinate *this, int y);
    void (*print)(coordinate *this);
    /* Data */
    int x;
    int y;
} coordinate;

/* Constructor */
coordinate *coordinate_create(void);
/* Destructor */
void coordinate_destroy(coordinate *this);
```

And now the implementing C file:

```
/* coordinates.c */

#include "coordinates.h"
#include <stdio.h>
#include <stdlib.h>

/* Constructor */
coordinate *coordinate_create(void)
{
    coordinate *c = malloc(sizeof(*c));
    if (c != 0)
    {
        c->setx = &coordinate_setx;
        c->sety = &coordinate_sety;
        c->print = &coordinate_print;
        c->x = 0;
        c->y = 0;
    }
    return c;
}

/* Destructor */
void coordinate_destroy(coordinate *this)
{
    if (this != NULL)
    {
        free(this);
    }
}

/* Methods */
static void coordinate_setx(coordinate *this, int x)
{
    if (this != NULL)
    {
        this->x = x;
    }
}

static void coordinate_sety(coordinate *this, int y)
{
    if (this != NULL)
    {
        this->y = y;
    }
}

static void coordinate_print(coordinate *this)
{
    if (this != NULL)
    {
        printf("Coordinate: (%i, %i)\n", this->x, this->y);
    }
    else
    {
        printf("NULL pointer exception!\n");
    }
}
```

An example usage of our coordinate class would be:

```
/* main.c */

#include "coordinates.h"
#include <stddef.h>

int main(void)
{
    /* Create and initialize pointers to coordinate objects */
    coordinate *c1 = coordinate_create();
    coordinate *c2 = coordinate_create();

    /* Now we can use our objects using our methods and passing the object as parameter */
    c1->setx(c1, 1);
    c1->sety(c1, 2);

    c2->setx(c2, 3);
    c2->sety(c2, 4);

    c1->print(c1);
    c2->print(c2);

    /* After using our objects we destroy them using our "destructor" function */
    coordinate_destroy(c1);
    c1 = NULL;
    coordinate_destroy(c2);
    c2 = NULL;

    return 0;
}
```

Section 13.6: Simple data structures

Structure data types are useful way to package related data and have them behave like a single variable.

Declaring a simple `struct` that holds two `int` members:

```
struct point
{
    int x;
    int y;
};
```

`x` and `y` are called the *members* (or *fields*) of `point` struct.

Defining and using structs:

```
struct point p;    // declare p as a point struct
p.x = 5;           // assign p member variables
p.y = 3;
```

Structs can be initialized at definition. The above is equivalent to:

```
struct point p = {5, 3};
```

Structs may also be initialized using designated initializers.

Accessing fields is also done using the `.` operator

```
printf("point is (x = %d, y = %d)", p.x, p.y);
```

Chapter 14: Standard Math

Section 14.1: Power functions - pow(), powf(), powl()

The following example code computes the sum of *1+4(3+3^2+3^3+3^4+...+3^N)* series using pow() family of standard math library.

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <fenv.h>

int main()
{
        double pwr, sum=0;
        int i, n;

        printf("\n1+4(3+3^2+3^3+3^4+...+3^N)=?\nEnter N:");
        scanf("%d",&n);
        if (n<=0) {
                printf("Invalid power N=%d", n);
                return -1;
        }

        for (i=0; i<n+1; i++) {
                errno = 0;
                feclearexcept(FE_ALL_EXCEPT);
                pwr = powl(3,i);
                if (fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW |
                        FE_UNDERFLOW)) {
                        perror("Math Error");
                }
                sum += i ? pwr : 0;
                printf("N= %d\tS= %g\n", i, 1+4*sum);
        }

        return 0;
}
```

Example Output:

```

1+4(3+3^2+3^3+3^4+...+3^N)=?
Enter N:10
N= 0    S= 1
N= 1    S= 13
N= 2    S= 49
N= 3    S= 157
N= 4    S= 481
N= 5    S= 1453
N= 6    S= 4369
N= 7    S= 13117
N= 8    S= 39361
N= 9    S= 118093
N= 10    S= 354289
```

Section 14.2: Double precision floating-point remainder: fmod()

This function returns the floating-point remainder of the division of `x/y`. The returned value has the same sign as x.

```
#include <math.h> /* for fmod() */
#include <stdio.h> /* for printf() */

int main(void)
{
    double x = 10.0;
    double y = 5.1;

    double modulus = fmod(x, y);

    printf("%lf\n", modulus); /* f is the same as lf. */

    return 0;
}
```

Output:

```
4.90000
```

Important: Use this function with care, as it can return unexpected values due to the operation of floating point values.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    printf("%f\n", fmod(1, 0.1));
    printf("%19.17f\n", fmod(1, 0.1));
    return 0;
}
```

Output:

```
0.1
0.09999999999999995
```

Section 14.3: Single precision and long double precision floating-point remainder: fmodf(), fmodl()

Version ≥ C99

These functions returns the floating-point remainder of the division of `x/y`. The returned value has the same sign as x.

Single Precision:

```
#include <math.h> /* for fmodf() */
#include <stdio.h> /* for printf() */
```

```c
int main(void)
{
    float x = 10.0;
    float y = 5.1;

    float modulus = fmodf(x, y);

    printf("%f\n", modulus); /* lf would do as well as modulus gets promoted to double. */
}
```

Output:

```
4.90000
```

Double Double Precision:

```c
#include <math.h> /* for fmodl() */
#include <stdio.h> /* for printf() */

int main(void)
{
    long double x = 10.0;
    long double y = 5.1;

    long double modulus = fmodl(x, y);

    printf("%Lf\n", modulus); /* Lf is for long double. */
}
```

Output:

```
4.90000
```

Chapter 15: Iteration Statements/Loops: for, while, do-while

Section 15.1: For loop

In order to execute a block of code over an over again, loops comes into the picture. The `for` loop is to be used when a block of code is to executed a fixed number of times. For example, in order to fill an array of size `n` with the user inputs, we need to execute `scanf()` for `n` times.

Version ≥ C99

```
#include <stddef.h>          // for size_t

int array[10];               // array of 10 int

for (size_t i = 0; i < 10; i++)    // i starts at 0 and finishes with 9
{
   scanf("%d", &array[i]);
}
```

In this way the `scanf()` function call is executed `n` times (10 times in our example), but is written only once.

Here, the variable `i` is the loop index, and it is best declared as presented. The type `size_t` (*size type*) should be used for everything that counts or loops through data objects.

This way of declaring variables inside the `for` is only available for compilers that have been updated to the C99 standard. If for some reason you are still stuck with an older compiler you can declare the loop index before the `for` loop:

Version < C99

```
#include <stddef.h>        /* for size_t */
size_t i;
int array[10];             /* array of 10 int */

for (i = 0; i < 10; i++)       /* i starts at 0 and finishes at 9 */
{
   scanf("%d", &array[i]);
}
```

Section 15.2: Loop Unrolling and Duff's Device

Sometimes, the straight forward loop cannot be entirely contained within the loop body. This is because, the loop needs to be primed by some statements ***B***. Then, the iteration begins with some statements ***A***, which are then followed by ***B*** again before looping.

```
do_B();
while (condition) {
    do_A();
    do_B();
}
```

To avoid potential cut/paste problems with repeating ***B*** twice in the code, Duff's Device could be applied to start the loop from the middle of the `while` body, using a switch statement and fall through behavior.

```
switch (true) while (condition) {
case false: do_A(); /* FALL THROUGH */
```

```c
default:    do_B(); /* FALL THROUGH */
}
```

Duff's Device was actually invented to implement loop unrolling. Imagine applying a mask to a block of memory, where `n` is a signed integral type with a positive value.

```c
do {
    *ptr++ ^= mask;
} while (--n > 0);
```

If `n` were always divisible by 4, you could unroll this easily as:

```c
do {
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
} while ((n -= 4) > 0);
```

But, with Duff's Device, the code can follow this unrolling idiom that jumps into the right place in the middle of the loop if `n` is not divisible by 4.

```c
switch (n % 4) do {
case 0: *ptr++ ^= mask; /* FALL THROUGH */
case 3: *ptr++ ^= mask; /* FALL THROUGH */
case 2: *ptr++ ^= mask; /* FALL THROUGH */
case 1: *ptr++ ^= mask; /* FALL THROUGH */
} while ((n -= 4) > 0);
```

This kind of manual unrolling is rarely required with modern compilers, since the compiler's optimization engine can unroll loops on the programmer's behalf.

Section 15.3: While loop

A `while` loop is used to execute a piece of code while a condition is true. The `while` loop is to be used when a block of code is to be executed a variable number of times. For example the code shown gets the user input, as long as the user inserts numbers which are not `0`. If the user inserts `0`, the while condition is not true anymore so execution will exit the loop and continue on to any subsequent code:

```c
int num = 1;

while (num != 0)
{
    scanf("%d", &num);
}
```

Section 15.4: Do-While loop

Unlike `for` and `while` loops, `do-while` loops check the truth of the condition at the end of the loop, which means the `do` block will execute once, and then check the condition of the `while` at the bottom of the block. Meaning that a `do-while` loop will *always* run at least once.

For example this `do-while` loop will get numbers from user, until the sum of these values is greater than or equal to `50`:

```
int num, sum;
num = sum = 0;

do
{
  scanf("%d", &num);
  sum += num;

} while (sum < 50);
```

`do-while` loops are relatively rare in most programming styles.

Section 15.5: Structure and flow of control in a for loop

```
for ([declaration-or-expression]; [expression2]; [expression3])
{
    /* body of the loop */
}
```

In a `for` loop, the loop condition has three expressions, all optional.

- The first expression, `declaration-or-expression`, *initializes* the loop. It is executed exactly once at the beginning of the loop.

Version ≥ C99

It can be either a declaration and initialization of a loop variable, or a general expression. If it is a declaration, the scope of the declared variable is restricted by the `for` statement.

Version < C99

Historical versions of C only allowed an expression, here, and the declaration of a loop variable had to be placed before the `for`.

- The second expression, `expression2`, is the *test condition*. It is first executed after the initialization. If the condition is **`true`**, then the control enters the body of the loop. If not, it shifts to outside the body of the loop at the end of the loop. Subsequently, this conditon is checked after each execution of the body as well as the update statement. When **`true`**, the control moves back to the beginning of the body of the loop. The condition is usually intended to be a check on the number of times the body of the loop executes. This is the primary way of exiting a loop, the other way being using jump statements.
- The third expression, `expression3`, is the *update statement*. It is executed after each execution of the body of the loop. It is often used to increment a variable keeping count of the number of times the loop body has executed, and this variable is called an *iterator*.

Each instance of execution of the loop body is called an *iteration*.

Example:

Version ≥ C99

```
for(int i = 0; i < 10 ; i++)
{
    printf("%d", i);
}
```

The output is:

```
0123456789
```

In the above example, first `i = 0` is executed, initializing `i`. Then, the condition `i < 10` is checked, which evaluates to be **`true`**. The control enters the body of the loop and the value of `i` is printed. Then, the control shifts to `i++`, updating the value of `i` from 0 to 1. Then, the condition is again checked, and the process continues. This goes on till the value of `i` becomes 10. Then, the condition `i < 10` evaluates **`false`**, after which the control moves out of the loop.

Section 15.6: Infinite Loops

A loop is said to be an *infinite loop* if the control enters but never leaves the body of the loop. This happens when the test condition of the loop never evaluates to **`false`**.

Example:

Version ≥ C99

```
for (int i = 0; i >= 0; )
{
    /* body of the loop where i is not changed*/
}
```

In the above example, the variable `i`, the iterator, is initialized to 0. The test condition is initially **`true`**. However, `i` is not modified anywhere in the body and the update expression is empty. Hence, `i` will remain 0, and the test condition will never evaluate to **`false`**, leading to an infinite loop.

Assuming that there are no jump statements, another way an infinite loop might be formed is by explicitly keeping the condition true:

```
while (true)
{
    /* body of the loop */
}
```

In a `for` loop, the condition statement optional. In this case, the condition is always **`true`** vacuously, leading to an infinite loop.

```
for (;;)
{
    /* body of the loop */
}
```

However, in certain cases, the condition might be kept **`true`** intentionally, with the intention of exiting the loop using a jump statement such as **`break`**.

```
while (true)
{
    /* statements */
    if (condition)
    {
         /* more statements */
         break;
    }
}
```

Chapter 16: Selection Statements

Section 16.1: if () Statements

One of the simplest ways to control program flow is by using `if` selection statements. Whether a block of code is to be executed or not to be executed can be decided by this statement.

The syntax for `if` selection statement in C could be as follows:

```
if(cond)
{
  statement(s);  /*to be executed, on condition being true*/
}
```

For example,

```
if (a > 1) {
    puts("a is larger than 1");
}
```

Where `a > 1` is a *condition* that has to evaluate to **`true`** in order to execute the statements inside the `if` block. In this example "a is larger than 1" is only printed if `a > 1` is true.

`if` selection statements can omit the wrapping braces `{` and `}` if there is only one statement within the block. The above example can be rewritten to

```
if (a > 1)
    puts("a is larger than 1");
```

However for executing multiple statements within block the braces have to used.

The *condition* for `if` can include multiple expressions. `if` will only perform the action if the end result of expression is true.

For example

```
if ((a > 1) && (b > 1)) {
    puts("a is larger than 1");
    a++;
}
```

will only execute the `printf` and `a++` if **both** `a` and `b` are greater than `1`.

Section 16.2: Nested if()...else VS if()..else Ladder

Nested `if()...else` statements take more execution time (they are slower) in comparison to an `if()...else` ladder because the nested `if()...else` statements check all the inner conditional statements once the outer conditional `if()` statement is satisfied, whereas the `if()..else` ladder will stop condition testing once any of the `if()` or the `else if()` conditional statements are true.

An `if()...else` ladder:

```
#include <stdio.h>

```

```c
int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if ((a < b) && (a < c))
  {
    printf("\na = %d is the smallest.", a);
  }
  else if ((b < a) && (b < c))
  {
    printf("\nb = %d is the smallest.", b);
  }
  else if ((c < a) && (c < b))
  {
    printf("\nc = %d is the smallest.", c);
  }
  else
  {
    printf("\nImprove your coding logic");
  }
  return 0;
}
```

Is, in the general case, considered to be better than the equivalent nested `if()...else`:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if (a < b)
  {
    if (a < c)
      {
        printf("\na = %d is the smallest.", a);
      }
    else
      {
        printf("\nc = %d is the smallest.", c);
      }
  }
  else
  {
    if(b < c)
    {
      printf("\nb = %d is the smallest.", b);
    }
    else
    {
      printf("\nc = %d is the smallest.", c);
    }
  }
  return 0;
}
```

Section 16.3: switch () Statements

`switch` statements are useful when you want to have your program do many different things according to the value of a particular test variable.

An example usage of `switch` statement is like this:

```
int a = 1;

switch (a) {
case 1:
    puts("a is 1");
    break;
case 2:
    puts("a is 2");
    break;
default:
    puts("a is neither 1 nor 2");
    break;
}
```

This example is equivalent to

```
int a = 1;

if (a == 1) {
    puts("a is 1");
} else if (a == 2) {
    puts("a is 2");
} else {
    puts("a is neither 1 nor 2");
}
```

If the value of `a` is 1 when the `switch` statement is used, `a is 1` will be printed. If the value of `a` is 2 then, `a is 2` will be printed. Otherwise, `a is neither 1 nor 2` will be printed.

`case n:` is used to describe where the execution flow will jump in when the value passed to `switch` statement is *n*. *n* must be compile-time constant and the same *n* can exist at most once in one `switch` statement.

`default:` is used to describe that when the value didn't match any of the choices for `case n:`. It is a good practice to include a `default` case in every switch statement to catch unexpected behavior.

A `break;` statement is required to jump out of the `switch` block.

Note: If you accidentally forget to add a `break` after the end of a `case`, the compiler will assume that you intend to "fall through" and all the subsequent case statements, if any, will be executed (unless a *break* statement is found in any of the subsequent cases), regardless of whether the subsequent case statement(s) match or not. This particular property is used to implement Duff's Device. This behavior is often considered a flaw in the C language specification.

Below is an example that shows effects of the absence of `break;`:

```
int a = 1;

switch (a) {
case 1:
case 2:
```

```
    puts("a is 1 or 2");
case 3:
    puts("a is 1, 2 or 3");
    break;
default:
    puts("a is neither 1, 2 nor 3");
    break;
}
```

When the value of a is 1 or 2, `a is 1 or 2` and `a is 1, 2 or 3` will both be printed. When a is 3, only `a is 1, 2 or 3` will be printed. Otherwise, `a is neither 1, 2 nor 3` will be printed.

Note that the `default` case is not necessary, especially when the set of values you get in the `switch` is finished and known at compile time.

The best example is using a `switch` on an **enum**.

```
enum msg_type { ACK, PING, ERROR };
void f(enum msg_type t)
{
  switch (t) {
  case ACK:
    // do nothing
    break;
  case PING:
    // do something
    break;
  case ERROR:
    // do something else
    break;
  }
}
```

There are multiple advantages of doing this:

- most compilers will report a warning if you don't handle a value (this would not be reported if a `default` case were present)
- for the same reason, if you add a new value to the **enum**, you will be notified of all the places where you forgot to handle the new value (with a `default` case, you would need to manually explore your code searching for such cases)
- The reader does not need to figure out "what is hidden by the `default:`", whether there other **enum** values or whether it is a protection for "just in case". And if there are other **enum** values, did the coder intentionally use the `default` case for them or is there a bug that was introduced when he added the value?
- handling each **enum** value makes the code self explanatory as you can't hide behind a wild card, you must explicitly handle each of them.

Nevertheless, you can't prevent someone to write evil code like:

```
enum msg_type t = (enum msg_type)666; // I'm evil
```

Thus you may add an extra check before your switch to detect it, if you really need it.

```
void f(enum msg_type t)
{
   if (!is_msg_type_valid(t)) {
      // Handle this unlikely error
   }
```

```
    switch(t) {
     // Same code than before
    }
}
```

Section 16.4: if () ... else statements and syntax

While `if` performs an action only when its condition evaluate to **`true`**, `if` / `else` allows you to specify the different actions when the condition **`true`** and when the condition is **`false`**.

Example:

```
if (a > 1)
    puts("a is larger than 1");
else
    puts("a is not larger than 1");
```

Just like the `if` statement, when the block within `if` or `else` is consisting of only one statement, then the braces can be omitted (but doing so is not recommended as it can easily introduce problems involuntarily). However if there's more than one statement within the `if` or `else` block, then the braces have to be used on that particular block.

```
if (a > 1)
{
    puts("a is larger than 1");
    a--;
}
else
{
    puts("a is not larger than 1");
    a++;
}
```

Section 16.5: if()...else Ladder Chaining two or more if () ... else statements

While the `if ()... else` statement allows to define only one (default) behaviour which occurs when the condition within the `if ()` is not met, chaining two or more `if () ... else` statements allow to define a couple more behaviours before going to the last `else` branch acting as a "default", if any.

Example:

```
int a = ... /* initialise to some value. */

if (a >= 1)
{
    printf("a is greater than or equals 1.\n");
}
else if (a == 0) //we already know that a is smaller than 1
{
    printf("a equals 0.\n");
}
else /* a is smaller than 1 and not equals 0, hence: */
{
    printf("a is negative.\n");
}
```

Chapter 17: Initialization

Section 17.1: Initialization of Variables in C

In the absence of explicit initialization, external and `static` variables are guaranteed to be initialized to zero; automatic variables (including `register` variables) have *indeterminate*1 (i.e., garbage) initial values.

Scalar variables may be initialized when they are defined by following the name with an equals sign and an expression:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

For external and `static` variables, the initializer must be a *constant expression*2; the initialization is done once, conceptually before the program begins execution.

For automatic and `register` variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls.

For example, see the code snippet below

```
int binsearch(int x, int v[], int n)
{
   int low = 0;
   int high = n - 1;
   int mid;
   ...
}
```

instead of

```
   int low, high, mid;

   low = 0;
   high = n - 1;
```

In effect, initialization of automatic variables are just shorthand for assignment statements. Which form to prefer is largely a matter of taste. We generally use explicit assignments, because initializers in declarations are harder to see and further away from the point of use. On the other hand, variables should only be declared when they're about to be used whenever possible.

Initializing an array:

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas.

For example, to initialize an array days with the number of days in each month:

```
int days_of_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

(Note that January is encoded as month zero in this structure.)

When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 12 in this case.

If there are fewer initializers for an array than the specified size, the others will be zero for all types of variables.

It is an error to have too many initializers. There is no standard way to specify repetition of an initializer — but GCC has an extension to do so.

Version < C99

In C89/C90 or earlier versions of C, there was no way to initialize an element in the middle of an array without supplying all the preceding values as well.

Version ≥ C99

With C99 and above, designated initializers allow you to initialize arbitrary elements of an array, leaving any uninitialized values as zeros.

Initializing Character arrays:

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

```
char chr_array[] = "hello";
```

is a shorthand for the longer but equivalent:

```
char chr_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

In this case, the array size is six (five characters plus the terminating `'\0'`).

1 What happens to a declared, uninitialized variable in C? Does it have a value?

2 Note that a *constant expression* is defined as something that can be evaluated at compile-time. So, `int global_var = f();` is invalid. Another common misconception is thinking of a `const` qualified variable as a *constant expression*. In C, `const` means "read-only", not "compile time constant". So, global definitions like `const int SIZE = 10; int global_arr[SIZE];` and `const int SIZE = 10; int global_var = SIZE;` are not legal in C.

Section 17.2: Using designated initializers

Version ≥ C99

C99 introduced the concept of *designated initializers.* These allow you to specify which elements of an array, structure or union are to be initialized by the values following.

Designated initializers for array elements

For a simple type like plain `int`:

```
int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 };
```

The term in square brackets, which can be any constant integer expression, specifies which element of the array is to be initialized by the value of the term after the `=` sign. Unspecified elements are default initialized, which means zeros are defined. The example shows the designated initializers in order; they do not have to be in order. The example shows gaps; those are legitimate. The example doesn't show two different initializations for the same element; that too is allowed (ISO/IEC 9899:2011, §6.7.9 Initialization, ¶19 *The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject*).

In this example, the size of the array is not defined explicitly, so the maximum index specified in the designated initializers dictates the size of the array — which would be 21 elements in the example. If the size was defined, initializing an entry beyond the end of the array would be an error, as usual.

Designated initializers for structures

You can specify which elements of a structure are initialized by using the `.element` notation:

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { .day = 4, .month = 7, .year = 1776 };
```

If elements are not listed, they are default initialized (zeroed).

Designated initializer for unions

You can specify which element of a union is initialize with a designated initializer.

Version = C89

Prior to the C standard, there was no way to initialize a `union`. The C89/C90 standard allows you to initialize the first member of a `union` — so the choice of which member is listed first matters.

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    } du;
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du = { .du_int = 1 } };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du = { .du_double = 3.14159 } };
```

Version ≥ C11

Note that C11 allows you to use anonymous union members inside a structure, so that you don't need the `du` name in the previous example:

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    };
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du_int = 1 };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du_double = 3.14159 };
```

Designated initializers for arrays of structures, etc

These constructs can be combined for arrays of structures containing elements that are arrays, etc. Using full sets of braces ensures that the notation is unambiguous.

```
typedef struct Date Date;  // See earlier in this example

struct date_range
{
    Date    dr_from;
    Date    dr_to;
    char    dr_what[80];
};

struct date_range ranges[] =
{
    [3] = { .dr_from = { .year = 1066, .month = 10, .day = 14 },
            .dr_to   = { .year = 1066, .month = 12, .day = 25 },
            .dr_what = "Battle of Hastings to Coronation of William the Conqueror",
          },
    [2] = { .dr_from = { .month = 7, .day =  4, .year = 1776 },
            .dr_to   = { .month = 5, .day = 14, .year = 1787 },
            .dr_what = "US Declaration of Independence to Constitutional Convention",
          }
 };
```

Specifying ranges in array initializers

GCC provides an extension that allows you to specify a range of elements in an array that should be given the same initializer:

```
int array[] = { [3 ... 7] = 29, 19 = 107 };
```

The triple dots need to be separate from the numbers lest one of the dots be interpreted as part of a floating point number (*maximimal munch* rule).

Section 17.3: Initializing structures and arrays of structures

Structures and arrays of structures can be initialized by a series of values enclosed in braces, one value per member of the structure.

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { 1776, 7, 4 };

struct Date uk_battles[] =
{
    { 1066, 10, 14 },    // Battle of Hastings
    { 1815,  6, 18 },    // Battle of Waterloo
    { 1805, 10, 21 },    // Battle of Trafalgar
};
```

Note that the array initialization could be written without the interior braces, and in times past (before 1990, say) often would have been written without them:

```c
struct Date uk_battles[] =
{
    1066, 10, 14,    // Battle of Hastings
    1815,  6, 18,    // Battle of Waterloo
    1805, 10, 21,    // Battle of Trafalgar
};
```

Although this works, it is not good modern style — you should not attempt to use this notation in new code and should fix the compiler warnings it usually yields.

See also designated initializers.

Chapter 18: Declaration vs Definition

Section 18.1: Understanding Declaration and Definition

A declaration introduces an identifier and describes its type, be it a type, object, or function. A declaration is what the compiler needs to accept references to that identifier. These are declarations:

```
extern int bar;
extern int g(int, int);
double f(int, double); /* extern can be omitted for function declarations */
double h1();           /* declaration without prototype */
double h2();           /* ditto                         */
```

A definition actually instantiates/implements this identifier. It's what the linker needs in order to link references to those entities. These are definitions corresponding to the above declarations:

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
double h1(int a, int b) {return -1.5;}
double h2() {}  /* prototype is implied in definition, same as double h2(void) */
```

A definition can be used in the place of a declaration.

However, it must be defined exactly once. If you forget to define something that's been declared and referenced somewhere, then the linker doesn't know what to link references to and complains about a missing symbols. If you define something more than once, then the linker doesn't know which of the definitions to link references to and complains about duplicated symbols.

Exception:

```
extern int i = 0;  /* defines i */
extern int j;  /* declares j */
```

This exception can be explained using concepts of "Strong symbols vs Weak symbols" (from a linker's perspective) . Please look here (Slide 22) for more explanation.

```
/* All are definitions. */
struct S { int a; int b; };             /* defines S */
struct X {                              /* defines X */
    int x;                              /* defines non-static data member x */
};
struct X anX;                                  /* defines anX */
```

Chapter 19: Command-line arguments

Parameter	Details
argc	argument count - initialized to the number of space-separated arguments given to the program from the command-line as well as the program name itself.
argv	argument vector - initialized to an array of `char`-pointers (strings) containing the arguments (and the program name) that was given on the command-line.

Section 19.1: Print the arguments to a program and convert to integer values

The following code will print the arguments to the program, and the code will attempt to convert each argument into a number (to a `long`):

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <limits.h>

int main(int argc, char* argv[]) {

    for (int i = 1; i < argc; i++) {
        printf("Argument %d is: %s\n", i, argv[i]);

        errno = 0;
        char *p;
        long argument_numValue = strtol(argv[i], &p, 10);

        if (p == argv[i]) {
            fprintf(stderr, "Argument %d is not a number.\n", i);
        }
        else if ((argument_numValue == LONG_MIN || argument_numValue == LONG_MAX) && errno ==
ERANGE) {
            fprintf(stderr, "Argument %d is out of range.\n", i);
        }
        else {
            printf("Argument %d is a number, and the value is: %ld\n",
                   i, argument_numValue);
        }
    }
    return 0;
}
```

References:

- strtol() returns an incorrect value
- Correct usage of strtol

Section 19.2: Printing the command line arguments

After receiving the arguments, you can print them as follows:

```
int main(int argc, char **argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("Argument %d: [%s]\n", i, argv[i]);
```

```
    }
}
```

Notes

1. The `argv` parameter can be also defined as `char *argv[]`.
2. `argv[0]` *may* contain the program name itself (depending on how the program was executed). The first "real" command line argument is at `argv[1]`, and this is the reason why the loop variable `i` is initialized to 1.
3. In the print statement, you can use `*(argv + i)` instead of `argv[i]` - it evaluates to the same thing, but is more verbose.
4. The square brackets around the argument value help identify the start and end. This can be invaluable if there are trailing blanks, newlines, carriage returns, or other oddball characters in the argument. Some variant on this program is a useful tool for debugging shell scripts where you need to understand what the argument list actually contains (although there are simple shell alternatives that are almost equivalent).

Section 19.3: Using GNU getopt tools

Command-line options for applications are not treated any differently from command-line arguments by the C language. They are just arguments which, in a Linux or Unix environment, traditionally begin with a dash (`-`).

With glibc in a Linux or Unix environment you can use the getopt tools to easily define, validate, and parse command-line options from the rest of your arguments.

These tools expect your options to be formatted according to the GNU Coding Standards, which is an extension of what POSIX specifies for the format of command-line options.

The example below demonstrates handling command-line options with the GNU getopt tools.

```
#include <stdio.h>
#include <getopt.h>
#include <string.h>

/* print a description of all supported options */
void usage (FILE *fp, const char *path)
{
    /* take only the last portion of the path */
    const char *basename = strrchr(path, '/');
    basename = basename ? basename + 1 : path;

    fprintf (fp, "usage: %s [OPTION]\n", basename);
    fprintf (fp, "  -h, --help\t\t"
                 "Print this help and exit.\n");
    fprintf (fp, "  -f, --file[=FILENAME]\t"
                 "Write all output to a file (defaults to out.txt).\n");
    fprintf (fp, "  -m, --msg=STRING\t"
                 "Output a particular message rather than 'Hello world'.\n");
}

/* parse command-line options and print message */
int main(int argc, char *argv[])
{
    /* for code brevity this example just uses fixed buffer sizes for strings */
    char filename[256] = { 0 };
    char message[256] = "Hello world";
    FILE *fp;
    int help_flag = 0;
    int opt;
```

```
    /* table of all supported options in their long form.
     * fields: name, has_arg, flag, val
     * `has_arg` specifies whether the associated long-form option can (or, in
     * some cases, must) have an argument. the valid values for `has_arg` are
     * `no_argument`, `optional_argument`, and `required_argument`.
     * if `flag` points to a variable, then the variable will be given a value
     * of `val` when the associated long-form option is present at the command
     * line.
     * if `flag` is NULL, then `val` is returned by `getopt_long` (see below)
     * when the associated long-form option is found amongst the command-line
     * arguments.
     */
    struct option longopts[] = {
        { "help", no_argument, &help_flag, 1 },
        { "file", optional_argument, NULL, 'f' },
        { "msg", required_argument, NULL, 'm' },
        { 0 }
    };

    /* infinite loop, to be broken when we are done parsing options */
    while (1) {
        /* getopt_long supports GNU-style full-word "long" options in addition
         * to the single-character "short" options which are supported by
         * getopt.
         * the third argument is a collection of supported short-form options.
         * these do not necessarily have to correlate to the long-form options.
         * one colon after an option indicates that it has an argument, two
         * indicates that the argument is optional. order is unimportant.
         */
        opt = getopt_long (argc, argv, "hf::m:", longopts, 0);

        if (opt == -1) {
            /* a return value of -1 indicates that there are no more options */
            break;
        }

        switch (opt) {
        case 'h':
            /* the help_flag and value are specified in the longopts table,
             * which means that when the --help option is specified (in its long
             * form), the help_flag variable will be automatically set.
             * however, the parser for short-form options does not support the
             * automatic setting of flags, so we still need this code to set the
             * help_flag manually when the -h option is specified.
             */
            help_flag = 1;
            break;
        case 'f':
            /* optarg is a global variable in getopt.h. it contains the argument
             * for this option. it is null if there was no argument.
             */
            printf ("outarg: '%s'\n", optarg);
            strncpy (filename, optarg ? optarg : "out.txt", sizeof (filename));
            /* strncpy does not fully guarantee null-termination */
            filename[sizeof (filename) - 1] = '\0';
            break;
        case 'm':
            /* since the argument for this option is required, getopt guarantees
             * that aptarg is non-null.
             */
            strncpy (message, optarg, sizeof (message));
            message[sizeof (message) - 1] = '\0';
```

```
            break;
        case '?':
            /* a return value of '?' indicates that an option was malformed.
             * this could mean that an unrecognized option was given, or that an
             * option which requires an argument did not include an argument.
             */
            usage (stderr, argv[0]);
            return 1;
        default:
            break;
        }
    }

    if (help_flag) {
        usage (stdout, argv[0]);
        return 0;
    }

    if (filename[0]) {
        fp = fopen (filename, "w");
    } else {
        fp = stdout;
    }

    if (!fp) {
        fprintf(stderr, "Failed to open file.\n");
        return 1;
    }

    fprintf (fp, "%s\n", message);

    fclose (fp);
    return 0;
}
```

It can be compiled with `gcc`:

```
gcc example.c -o example
```

It supports three command-line options (`--help`, `--file`, and `--msg`). All have a "short form" as well (`-h`, `-f`, and `-m`). The "file" and "msg" options both accept arguments. If you specify the "msg" option, its argument is required.

Arguments for options are formatted as:

- `--option=value` (for long-form options)
- `-ovalue` or `-o"value"` (for short-form options)

Chapter 20: Files and I/O streams

Parameter	Details
const char *mode	A string describing the opening mode of the file-backed stream. See remarks for possible values.
int whence	Can be `SEEK_SET` to set from the beginning of the file, `SEEK_END` to set from its end, or `SEEK_CUR` to set relative to the current cursor value. Note: `SEEK_END` is non-portable.

Section 20.1: Open and write to file

```
#include <stdio.h>   /* for perror(), fopen(), fputs() and fclose() */
#include <stdlib.h>  /* for the EXIT_* macros */

int main(int argc, char **argv)
{
    int e = EXIT_SUCCESS;

    /* Get path from argument to main else default to output.txt */
    char *path = (argc > 1) ? argv[1] : "output.txt";

    /* Open file for writing and obtain file pointer */
    FILE *file = fopen(path, "w");

    /* Print error message and exit if fopen() failed */
    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Writes text to file. Unlike puts(), fputs() does not add a new-line. */
    if (fputs("Output in file.\n", file) == EOF)
    {
        perror(path);
        e = EXIT_FAILURE;
    }

    /* Close file */
    if (fclose(file))
    {
        perror(path);
        return EXIT_FAILURE;
    }
    return e;
}
```

This program opens the file with name given in the argument to main, defaulting to `output.txt` if no argument is given. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file. If the files does not already exist the `fopen()` call creates it.

If the `fopen()` call fails for some reason, it returns a `NULL` value and sets the global `errno` variable value. This means that the program can test the returned value after the `fopen()` call and use `perror()` if `fopen()` fails.

If the `fopen()` call succeeds, it returns a valid `FILE` pointer. This pointer can then be used to reference this file until `fclose()` is called on it.

The `fputs()` function writes the given text to the opened file, replacing any previous contents of the file. Similarly to `fopen()`, the `fputs()` function also sets the `errno` value if it fails, though in this case the function returns `EOF` to

indicate the fail (it otherwise returns a non-negative value).

The `fclose()` function flushes any buffers, closes the file and frees the memory pointed to by `FILE *`. The return value indicates completion just as `fputs()` does (though it returns '0' if successful), again also setting the `errno` value in the case of a fail.

Section 20.2: Run process

```
#include <stdio.h>

void print_all(FILE *stream)
{
    int c;
    while ((c = getc(stream)) != EOF)
        putchar(c);
}
int main(void)
{
    FILE *stream;

    /* call netstat command. netstat is available for Windows and Linux */
    if ((stream = popen("netstat", "r")) == NULL)
        return 1;

    print_all(stream);
    pclose(stream);
    return 0;
}
```

This program runs a process (`netstat`) via `popen()` and reads all the standard output from the process and echoes that to standard output.

Note: `popen()` does not exist in the standard C library, but it is rather a part of POSIX C)

Section 20.3: fprintf

You can use `fprintf` on a file just like you might on a console with `printf`. For example to keep track of game wins, losses and ties you might write

```
/* saves wins, losses and, ties */
void savewlt(FILE *fout, int wins, int losses, int ties)
{
    fprintf(fout, "Wins: %d\nTies: %d\nLosses: %d\n", wins, ties, losses);
}
```

A side note: Some systems (infamously, Windows) do not use what most programmers would call "normal" line endings. While UNIX-like systems use \n to terminate lines, Windows uses a pair of characters: \r (carriage return) and \n (line feed). This sequence is commonly called CRLF. However, whenever using C, you do not need to worry about these highly platform-dependent details. A C compiler is required to convert every instance of \n to the correct platform line ending. So a Windows compiler would convert \n to \r\n, but a UNIX compiler would keep it as-is.

Section 20.4: Get lines from a file using getline()

The POSIX C library defines the `getline()` function. This function allocates a buffer to hold the line contents and returns the new line, the number of characters in the line, and the size of the buffer.

Example program that gets each line from `example.txt`:

```
#include <stdlib.h>
#include <stdio.h>


#define FILENAME "example.txt"

int main(void)
{
  /* Open the file for reading */
  char *line_buf = NULL;
  size_t line_buf_size = 0;
  int line_count = 0;
  ssize_t line_size;
  FILE *fp = fopen(FILENAME, "r");
  if (!fp)
  {
    fprintf(stderr, "Error opening file '%s'\n", FILENAME);
    return EXIT_FAILURE;
  }

  /* Get the first line of the file. */
  line_size = getline(&line_buf, &line_buf_size, fp);

  /* Loop through until we are done with the file. */
  while (line_size >= 0)
  {
    /* Increment our line count */
    line_count++;

    /* Show the line details */
    printf("line[%06d]: chars=%06zd, buf size=%06zu, contents: %s", line_count,
        line_size, line_buf_size, line_buf);

    /* Get the next line */
    line_size = getline(&line_buf, &line_buf_size, fp);
  }

  /* Free the allocated line buffer */
  free(line_buf);
  line_buf = NULL;

  /* Close the file now that we are done with it */
  fclose(fp);

  return EXIT_SUCCESS;
}
```

Input file `example.txt`

```
This is a file
which has
multiple lines
with various indentation,
blank lines



a really long line to show that getline() will reallocate the line buffer if the length of a line
is too long to fit in the buffer it has been given,
and punctuation at the end of the lines.
```

Output

```
line[000001]: chars=000015, buf size=000016, contents: This is a file
line[000002]: chars=000012, buf size=000016, contents:   which has
line[000003]: chars=000015, buf size=000016, contents: multiple lines
line[000004]: chars=000030, buf size=000032, contents:     with various indentation,
line[000005]: chars=000012, buf size=000032, contents: blank lines
line[000006]: chars=000001, buf size=000032, contents:
line[000007]: chars=000001, buf size=000032, contents:
line[000008]: chars=000001, buf size=000032, contents:
line[000009]: chars=000150, buf size=000160, contents: a really long line to show that getline()
will reallocate the line buffer if the length of a line is too long to fit in the buffer it has
been given,
line[000010]: chars=000042, buf size=000160, contents:  and punctuation at the end of the lines.
line[000011]: chars=000001, buf size=000160, contents:
```

In the example, `getline()` is initially called with no buffer allocated. During this first call, `getline()` allocates a buffer, reads the first line and places the line's contents in the new buffer. On subsequent calls, `getline()` updates the same buffer and only reallocates the buffer when it is no longer large enough to fit the whole line. The temporary buffer is then freed when we are done with the file.

Another option is `getdelim()`. This is the same as `getline()` except you specify the line ending character. This is only necessary if the last character of the line for your file type is not '\n'. `getline()` works even with Windows text files because with the multibyte line ending (`"\r\n"`)'\n'` is still the last character on the line.

Example implementation of `getline()`

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <stdint.h>


#if !(defined _POSIX_C_SOURCE)
typedef long int ssize_t;
#endif

/* Only include our version of getline() if the POSIX version isn't available. */

#if !(defined _POSIX_C_SOURCE) || _POSIX_C_SOURCE < 200809L

#if !(defined SSIZE_MAX)
#define SSIZE_MAX (SIZE_MAX >> 1)
#endif

ssize_t getline(char **pline_buf, size_t *pn, FILE *fin)
{
  const size_t INITALLOC = 16;
  const size_t ALLOCSTEP = 16;
  size_t num_read = 0;

  /* First check that none of our input pointers are NULL. */
  if ((NULL == pline_buf) || (NULL == pn) || (NULL == fin))
  {
    errno = EINVAL;
    return -1;
  }

  /* If output buffer is NULL, then allocate a buffer. */
  if (NULL == *pline_buf)
```

```
{
  *pline_buf = malloc(INITALLOC);
  if (NULL == *pline_buf)
  {
    /* Can't allocate memory. */
    return -1;
  }
  else
  {
    /* Note how big the buffer is at this time. */
    *pn = INITALLOC;
  }
}

/* Step through the file, pulling characters until either a newline or EOF. */

{
  int c;
  while (EOF != (c = getc(fin)))
  {
    /* Note we read a character. */
    num_read++;

    /* Reallocate the buffer if we need more room */
    if (num_read >= *pn)
    {
      size_t n_realloc = *pn + ALLOCSTEP;
      char * tmp = realloc(*pline_buf, n_realloc + 1); /* +1 for the trailing NUL. */
      if (NULL != tmp)
      {
        /* Use the new buffer and note the new buffer size. */
        *pline_buf = tmp;
        *pn = n_realloc;
      }
      else
      {
        /* Exit with error and let the caller free the buffer. */
        return -1;
      }

      /* Test for overflow. */
      if (SSIZE_MAX < *pn)
      {
        errno = ERANGE;
        return -1;
      }
    }

    /* Add the character to the buffer. */
    (*pline_buf)[num_read - 1] = (char) c;

    /* Break from the loop if we hit the ending character. */
    if (c == '\n')
    {
      break;
    }
  }

  /* Note if we hit EOF. */
  if (EOF == c)
  {
    errno = 0;
```

```
        return -1;
      }
    }

    /* Terminate the string by suffixing NUL. */
    (*pline_buf)[num_read] = '\0';

    return (ssize_t) num_read;
}

#endif
```

Section 20.5: fscanf()

Let's say we have a text file and we want to read all words in that file, in order to do some requirements.

file.txt:

```
This is just
a test file
to be used by fscanf()
```

This is the main function:

```
#include <stdlib.h>
#include <stdio.h>

void printAllWords(FILE *);

int main(void)
{
  FILE *fp;

  if ((fp = fopen("file.txt", "r")) == NULL) {
      perror("Error opening file");
      exit(EXIT_FAILURE);
  }

  printAllWords(fp);

  fclose(fp);

  return EXIT_SUCCESS;
}

void printAllWords(FILE * fp)
{
    char tmp[20];
    int i = 1;

    while (fscanf(fp, "%19s", tmp) != EOF) {
        printf("Word %d: %s\n", i, tmp);
        i++;
    }
}
```

The output will be:

```
Word 1: This
Word 2: is
Word 3: just
Word 4: a
Word 5: test
Word 6: file
Word 7: to
Word 8: be
Word 9: used
Word 10: by
Word 11: fscanf()
```

Section 20.6: Read lines from a file

The `stdio.h` header defines the `fgets()` function. This function reads a line from a stream and stores it in a specified string. The function stops reading text from the stream when either `n - 1` characters are read, the newline character (`'\n'`) is read or the end of file (EOF) is reached.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LENGTH 80

int main(int argc, char **argv)
{
    char *path;
    char line[MAX_LINE_LENGTH] = {0};
    unsigned int line_count = 0;

    if (argc < 1)
        return EXIT_FAILURE;
    path = argv[1];

    /* Open file */
    FILE *file = fopen(path, "r");

    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Get each line until there are none left */
    while (fgets(line, MAX_LINE_LENGTH, file))
    {
        /* Print each line */
        printf("line[%06d]: %s", ++line_count, line);

        /* Add a trailing newline to lines that don't already have one */
        if (line[strlen(line) - 1] != '\n')
            printf("\n");
    }

    /* Close file */
    if (fclose(file))
    {
        return EXIT_FAILURE;
        perror(path);
```

```
    }
}
```

Calling the program with an argument that is a path to a file containing the following text:

```
This is a file
which has
multiple lines
with various indentation,
blank lines



a really long line to show that the line will be counted as two lines if the length of a line is
too long to fit in the buffer it has been given,
and punctuation at the end of the lines.
```

Will result in the following output:

```
line[000001]: This is a file
line[000002]:   which has
line[000003]: multiple lines
line[000004]:     with various indentation,
line[000005]: blank lines
line[000006]:
line[000007]:
line[000008]:
line[000009]: a really long line to show that the line will be counted as two lines if the le
line[000010]: ngth of a line is too long to fit in the buffer it has been given,
line[000011]:  and punctuation at the end of the lines.
line[000012]:
```

This very simple example allows a fixed maximum line length, such that longer lines will effectively be counted as two lines. The `fgets()` function requires that the calling code provide the memory to be used as the destination for the line that is read.

POSIX makes the `getline()` function available which instead internally allocates memory to enlarge the buffer as necessary for a line of any length (as long as there is sufficient memory).

Section 20.7: Open and write to a binary file

```
#include <stdlib.h>
#include <stdio.h>


int main(void)
{
   result = EXIT_SUCCESS;

   char file_name[] = "outbut.bin";
   char str[] = "This is a binary file example";
   FILE * fp = fopen(file_name, "wb");

   if (fp == NULL)  /* If an error occurs during the file creation */
   {
     result = EXIT_FAILURE;
     fprintf(stderr, "fopen() failed for '%s'\n", file_name);
```

```c
    }
    else
    {
      size_t element_size = sizeof *str;
      size_t elements_to_write = sizeof str;

      /* Writes str (_including_ the NUL-terminator) to the binary file. */
      size_t elements_written = fwrite(str, element_size, elements_to_write, fp);
      if (elements_written != elements_to_write)
      {
        result = EXIT_FAILURE;
        /* This works for >=c99 only, else the z length modifier is unknown. */
        fprintf(stderr, "fwrite() failed: wrote only %zu out of %zu elements.\n",
          elements_written, elements_to_write);
        /* Use this for <c99: *
        fprintf(stderr, "fwrite() failed: wrote only %lu out of %lu elements.\n",
          (unsigned long) elements_written, (unsigned long) elements_to_write);
         */
      }

      fclose(fp);
    }

    return result;
}
```

This program creates and writes text in the binary form through the `fwrite` function to the file `output.bin`.

If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.

A binary stream is an ordered sequence of characters that can transparently record internal data. In this mode, bytes are written between the program and the file without any interpretation.

To write integers portably, it must be known whether the file format expects them in big or little-endian format, and the size (usually 16, 32 or 64 bits). Bit shifting and masking may then be used to write out the bytes in the correct order. Integers in C are not guaranteed to have two's complement representation (though almost all implementations do). Fortunately a conversion to unsigned *is* guaranteed to use twos complement. The code for writing a signed integer to a binary file is therefore a little surprising.

```c
/* write a 16-bit little endian integer */
int fput16le(int x, FILE *fp)
{
    unsigned int rep = x;
    int e1, e2;

    e1 = fputc(rep & 0xFF, fp);
    e2 = fputc((rep >> 8) & 0xFF, fp);

    if(e1 == EOF || e2 == EOF)
        return EOF;
    return 0;
}
```

The other functions follow the same pattern with minor modifications for size and byte order.

Chapter 21: Formatted Input/Output

Section 21.1: Conversion Specifiers for printing

Conversion Specifier	Type of Argument	Description
`i`, `d`	int	prints decimal
`u`	unsigned int	prints decimal
`o`	unsigned int	prints octal
`x`	unsigned int	prints hexadecimal, lower-case
`X`	unsigned int	prints hexadecimal, upper-case
`f`	double	prints float with a default precision of 6, if no precision is given (lower-case used for special numbers `nan` and `inf` or `infinity`)
`F`	double	prints float with a default precision of 6, if no precision is given (upper-case used for special numbers `NAN` and `INF` or `INFINITY`)
`e`	double	prints float with a default precision of 6, if no precision is given, using scientific notation using mantissa/exponent; lower-case exponent and special numbers
`E`	double	prints float with a default precision of 6, if no precision is given, using scientific notation using mantissa/exponent; upper-case exponent and special numbers
`g`	double	uses either `f` or `e` *[see below]*
`G`	double	uses either `F` or `E` *[see below]*
`a`	double	prints hexadecimal, lower-case
`A`	double	prints hexadecimal, upper-case
`c`	char	prints single character
`s`	char*	prints string of characters up to a `NUL` terminator, or truncated to length given by precision, if specified
`p`	void*	prints `void`-pointer value; a non`void`-pointer should be explicitly converted ("cast") to `void*`; pointer to object only, **not** a function-pointer
`%`	n/a	prints the `%` character
`n`	int *	***write*** the number of bytes printed so far into the `int` pointed at.

Note that length modifiers can be applied to `%n` (e.g. `%hhn` indicates that *a following `n` conversion specifier applies to a pointer to a `signed char` argument*, according to the ISO/IEC 9899:2011 §7.21.6.1 ¶7).

Note that the floating point conversions apply to types `float` and `double` because of default promotion rules — §6.5.2.2 Function calls, ¶7 *The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.*) Thus, functions such as `printf()` are only ever passed `double` values, even if the variable referenced is of type `float`.

With the `g` and `G` formats, the choice between `e` and `f` (or `E` and `F`) notation is documented in the C standard and in the POSIX specification for [`printf()`]:

> The double argument representing a floating-point number shall be converted in the style `f` or `e` (or in the style `F` or `E` in the case of a `G` conversion specifier), depending on the value converted and the precision. Let `P` equal the precision if non-zero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style `E` would have an exponent of `X`:

> - If P > X >= -4, the conversion shall be with style `f` (or `F`) and precision `P - (X+1)`.
> - Otherwise, the conversion shall be with style `e` (or `E`) and precision `P - 1`.

> Finally, unless the '#' flag is used, any trailing zeros shall be removed from the fractional portion of the result and the decimal-point character shall be removed if there is no fractional portion remaining.

Section 21.2: The printf() Function

Accessed through including `<stdio.h>`, the function `printf()` is the primary tool used for printing text to the console in C.

```
printf("Hello world!");
// Hello world!
```

Normal, unformatted character arrays can be printed by themselves by placing them directly in between the parentheses.

```
printf("%d is the answer to life, the universe, and everything.", 42);
// 42 is the answer to life, the universe, and everything.

int x = 3;
char y = 'Z';
char* z = "Example";
printf("Int: %d, Char: %c, String: %s", x, y, z);
// Int: 3, Char: Z, String: Example
```

Alternatively, integers, floating-point numbers, characters, and more can be printed using the escape character `%`, followed by a character or sequence of characters denoting the format, known as the *format specifier*.

All additional arguments to the function `printf()` are separated by commas, and these arguments should be in the same order as the format specifiers. Additional arguments are ignored, while incorrectly typed arguments or a lack of arguments will cause errors or undefined behavior. Each argument can be either a literal value or a variable.

After successful execution, the number of characters printed is returned with type `int`. Otherwise, a failure returns a negative value.

Section 21.3: Printing format flags

The C standard (C11, and C99 too) defines the following flags for `printf()`:

Flag	Conversions	Meaning
`-`	all	The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.
`+`	signed numeric	The result of a signed conversion shall always begin with a sign ('+' or '-'). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.
`<space>`	signed numeric	If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a `<space>` shall be prefixed to the result. This means that if the `<space>` and '+' flags both appear, the `<space>` flag shall be ignored.
`#`	all	Specifies that the value is to be converted to an alternative form. For `o` conversion, it shall increase the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For `x` or `X` conversion specifiers, a non-zero result shall have `0x` (or `0X`) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For `g` and `G` conversion specifiers, trailing zeros shall not be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

`0`	numeric	For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the '0' and '-' flags both appear, the '0' flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag shall be ignored. ⌦ If the '0' and **`<apostrophe>`** flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined. ⌫

These flags are also supported by Microsoft with the same meanings.

The POSIX specification for `printf()` adds:

Flag	Conversions	Meaning
`'`	i, d, u, f, F, g, G	The integer portion of the result of a decimal conversion shall be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.

Section 21.4: Printing the Value of a Pointer to an Object

To print the value of a pointer to an object (as opposed to a function pointer) use the `p` conversion specifier. It is defined to print `void`-pointers only, so to print out the value of a non `void`-pointer it needs to be explicitly converted ("casted*") to `void*`.

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>  /* for printf() */

int main(void)
{
  int i;
  int * p = &i;

  printf("The address of i is %p.\n", (void*) p);

  return EXIT_SUCCESS;
}
```

Version ≥ C99

Using `<inttypes.h>` and `uintptr_t`

Another way to print pointers in C99 or later uses the `uintptr_t` type and the macros from **`<inttypes.h>`**:

```
#include <inttypes.h> /* for uintptr_t and PRIXPTR */
#include <stdio.h>    /* for printf() */

int main(void)
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%" PRIXPTR ".\n", (uintptr_t)p);

  return 0;
}
```

In theory, there might not be an integer type that can hold any pointer converted to an integer (so the type `uintptr_t` might not exist). In practice, it does exist. Pointers to functions need not be convertible to the `uintptr_t` type — though again they most often are convertible.

If the `uintptr_t` type exists, so does the `intptr_t` type. It is not clear why you'd ever want to treat addresses as

signed integers, though.

Version = K&R Version < C89

Pre-Standard History:

Prior to C89 during K&R-C times there was no type `void*` (nor header **`<stdlib.h>`**, nor prototypes, and hence no `int main(void)` notation), so the pointer was cast to `long unsigned int` and printed using the `lx` length modifier/conversion specifier.

The example below is just for informational purpose. Nowadays this is invalid code, which very well might provoke the infamous Undefined Behaviour.

```
#include <stdio.h> /* optional in pre-standard C - for printf() */

int main()
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%lx.\n", (long unsigned) p);

  return 0;
}
```

Section 21.5: Printing the Difference of the Values of two Pointers to an Object

Subtracting the values of two pointers to an object results in a signed integer *1. So it would be printed using *at least* the `d` conversion specifier.

To make sure there is a type being wide enough to hold such a "pointer-difference", since C99 **`<stddef.h>`** defines the type `ptrdiff_t`. To print a `ptrdiff_t` use the `t` length modifier.

Version ≥ C99

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */
#include <stddef.h> /* for ptrdiff_t */


int main(void)
{
  int a[2];
  int * p1 = &a[0], * p2 = &a[1];
  ptrdiff_t pd = p2 - p1;

  printf("p1 = %p\n", (void*) p1);
  printf("p2 = %p\n", (void*) p2);
  printf("p2 - p1 = %td\n", pd);

  return EXIT_SUCCESS;
}
```

The result might look like this:

```
p1 = 0x7fff6679f430
p2 = 0x7fff6679f434
p2 - p1 = 1
```

Please note that the resulting value of the difference is scaled by the size of the type the pointers subtracted point to, an `int` here. The size of an `int` for this example is 4.

*1 If the two pointers to be subtracted do not point to the same object the behaviour is undefined.

Section 21.6: Length modifiers

The C99 and C11 standards specify the following length modifiers for `printf()`; their meanings are:

Modifier	Modifies	Applies to
hh	d, i, o, u, x, or X	`char`, `signed char` or `unsigned char`
h	d, i, o, u, x, or X	`short int` or `unsigned short int`
l	d, i, o, u, x, or X	`long int` or `unsigned long int`
l	a, A, e, E, f, F, g, or G	`double` (for compatibility with `scanf()`; undefined in C90)
ll	d, i, o, u, x, or X	`long long int` or `unsigned long long int`
j	d, i, o, u, x, or X	`intmax_t` or `uintmax_t`
z	d, i, o, u, x, or X	`size_t` or the corresponding signed type (`ssize_t` in POSIX)
t	d, i, o, u, x, or X	`ptrdiff_t` or the corresponding unsigned integer type
L	a, A, e, E, f, F, g, or G	`long double`

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

Microsoft specifies some different length modifiers, and explicitly does not support `hh`, `j`, `z`, or `t`.

Modifier	Modifies	Applies to
I32	d, i, o, x, or X	`__int32`
I32	o, u, x, or X	`unsigned __int32`
I64	d, i, o, x, or X	`__int64`
I64	o, u, x, or X	`unsigned __int64`
I	d, i, o, x, or X	`ptrdiff_t` (that is, `__int32` on 32-bit platforms, `__int64` on 64-bit platforms)
I	o, u, x, or X	`size_t` (that is, `unsigned __int32` on 32-bit platforms, `unsigned __int64` on 64-bit platforms)
l or L	a, A, e, E, f, g, or G	`long double` (In Visual C++, although `long double` is a distinct type, it has the same internal representation as `double`.)
l or w	c or C	Wide character with `printf` and `wprintf` functions. (An `lc`, `lC`, `wc` or `wC` type specifier is synonymous with `C` in `printf` functions and with `c` in `wprintf` functions.)
l or w	s, S, or Z	Wide-character string with `printf` and `wprintf` functions. (An `ls`, `lS`, `ws` or `wS` type specifier is synonymous with `S` in `printf` functions and with `s` in `wprintf` functions.)

Note that the `C`, `S`, and `Z` conversion specifiers and the `I`, `I32`, `I64`, and `w` length modifiers are Microsoft extensions. Treating `l` as a modifier for `long double` rather than `double` is different from the standard, though you'll be hard-pressed to spot the difference unless `long double` has a different representation from `double`.

Chapter 22: Pointers

A pointer is a type of variable which can store the address of another object or a function.

Section 22.1: Introduction

A pointer is declared much like any other variable, except an asterisk (`*`) is placed between the type and the name of the variable to denote it is a pointer.

```
int *pointer; /* inside a function, pointer is uninitialized and doesn't point to any valid object
yet */
```

To declare two pointer variables of the same type, in the same declaration, use the asterisk symbol before each identifier. For example,

```
int *iptr1, *iptr2;
int *iptr3,  iptr4;  /* iptr3 is a pointer variable, whereas iptr4 is misnamed and is an int */
```

The address-of or reference operator denoted by an ampersand (`&`) gives the address of a given variable which can be placed in a pointer of appropriate type.

```
int value = 1;
pointer = &value;
```

The indirection or dereference operator denoted by an asterisk (`*`) gets the contents of an object pointed to by a pointer.

```
printf("Value of pointed to integer: %d\n", *pointer);
/* Value of pointed to integer: 1 */
```

If the pointer points to a structure or union type then you can dereference it and access its members directly using the `->` operator:

```
SomeStruct *s = &someObject;
s->someMember = 5; /* Equivalent to (*s).someMember = 5 */
```

In C, a pointer is a distinct value type which can be reassigned and otherwise is treated as a variable in its own right. For example the following example prints the value of the pointer (variable) itself.

```
printf("Value of the pointer itself: %p\n", (void *)pointer);
/* Value of the pointer itself: 0x7ffcd41b06e4 */
/* This address will be different each time the program is executed */
```

Because a pointer is a mutable variable, it is possible for it to not point to a valid object, either by being set to null

```
pointer = 0;     /* or alternatively */
pointer = NULL;
```

or simply by containing an arbitrary bit pattern that isn't a valid address. The latter is a very bad situation, because it cannot be tested before the pointer is being dereferenced, there is only a test for the case a pointer is null:

```
if (!pointer) exit(EXIT_FAILURE);
```

A pointer may only be dereferenced if it points to a *valid* object, otherwise the behavior is undefined. Many modern implementations may help you by raising some kind of error such as a segmentation fault and terminate execution, but others may just leave your program in an invalid state.

The value returned by the dereference operator is a mutable alias to the original variable, so it can be changed, modifying the original variable.

```
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", *pointer);
/* Value of pointed to variable after change: 2 */
```

Pointers are also re-assignable. This means that a pointer pointing to an object can later be used to point to another object of the same type.

```
int value2 = 10;
pointer = &value2;
printf("Value from pointer: %d\n", *pointer);
/* Value from pointer: 10 */
```

Like any other variable, pointers have a specific type. You can't assign the address of a `short int` to a pointer to a `long int`, for instance. Such behavior is referred to as type punning and is forbidden in C, though there are a few exceptions.

Although pointer must be of a specific type, the memory allocated for each type of pointer is equal to the memory used by the environment to store addresses, rather than the size of the type that is pointed to.

```
#include <stdio.h>

int main(void) {
    printf("Size of int pointer: %zu\n", sizeof (int*));      /* size 4 bytes */
    printf("Size of int variable: %zu\n", sizeof (int));      /* size 4 bytes */
    printf("Size of char pointer: %zu\n", sizeof (char*));    /* size 4 bytes */
    printf("Size of char variable: %zu\n", sizeof (char));    /* size 1 bytes */
    printf("Size of short pointer: %zu\n", sizeof (short*));  /* size 4 bytes */
    printf("Size of short variable: %zu\n", sizeof (short));  /* size 2 bytes */
    return 0;
}
```

(NB: if you are using Microsoft Visual Studio, which does not support the C99 or C11 standards, you must use `%Iu`1 instead of `%zu` in the above sample.)

Note that the results above can vary from environment to environment in numbers but all environments would show equal sizes for different types of pointer.

Extract based on information from Cardiff University C Pointers Introduction

Pointers and Arrays

Pointers and arrays are intimately connected in C. Arrays in C are always held in contiguous locations in memory. Pointer arithmetic is always scaled by the size of the item pointed to. So if we have an array of three doubles, and a pointer to the base, `*ptr` refers to the first double, `*(ptr + 1)` to the second, `*(ptr + 2)` to the third. A more convenient notation is to use array notation `[]`.

```
double point[3] = {0.0, 1.0, 2.0};
double *ptr = point;
```

```c
/* prints x 0.0, y 1.0 z 2.0 */
printf("x %f y %f z %f\n", ptr[0], ptr[1], ptr[2]);
```

So essentially ptr and the array name are interchangeable. This rule also means that an array decays to a pointer when passed to a subroutine.

```c
double point[3] = {0.0, 1.0, 2.0};

printf("length of point is %s\n", length(point));

/* get the distance of a 3D point from the origin */
double length(double *pt)
{
   return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2])
}
```

A pointer may point to any element in an array, or to the element beyond the last element. It is however an error to set a pointer to any other value, including the element before the array. (The reason is that on segmented architectures the address before the first element may cross a segment boundary, the compiler ensures that does not happen for the last element plus one).

Footnote 1: Microsoft format information can be found via `printf()` and format specification syntax.

Section 22.2: Common errors

Improper use of pointers are frequently a source of bugs that can include security bugs or program crashes, most often due to segmentation faults.

Not checking for allocation failures

Memory allocation is not guaranteed to succeed, and may instead return a `NULL` pointer. Using the returned value, without checking if the allocation is successful, invokes undefined behavior. This usually leads to a crash, but there is no guarantee that a crash will happen so relying on that can also lead to problems.

For example, unsafe way:

```c
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* UNSAFE, because s might be a null pointer */
```

Safe way:

```c
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* This is safe, we have checked that s is valid */
}
```

Using literal numbers instead of sizeof when requesting memory

For a given compiler/machine configuration, types have a known size; however, there isn't any standard which defines that the size of a given type (other than `char`) will be the same for all compiler/machine configurations. If the code uses 4 instead of `sizeof(int)` for memory allocation, it may work on the original machine, but the code isn't necessarily portable to other machines or compilers. Fixed sizes for types should be replaced by

sizeof(that_type) or sizeof(*var_ptr_to_that_type).

Non-portable allocation:

```
int *intPtr = malloc(4*1000);    /* allocating storage for 1000 int */
long *longPtr = malloc(8*1000);  /* allocating storage for 1000 long */
```

Portable allocation:

```
int *intPtr = malloc(sizeof(int)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(long)*1000);  /* allocating storage for 1000 long */
```

Or, better still:

```
int *intPtr = malloc(sizeof(*intPtr)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(*longPtr)*1000);  /* allocating storage for 1000 long */
```

Memory leaks

Failure to de-allocate memory using `free` leads to a buildup of non-reusable memory, which is no longer used by the program; this is called a memory leak. Memory leaks waste memory resources and can lead to allocation failures.

Logical errors

All allocations must follow the same pattern:

1. Allocation using `malloc` (or `calloc`)
2. Usage to store data
3. De-allocation using `free`

Failure to adhere to this pattern, such as using memory after a call to `free` (dangling pointer) or before a call to `malloc` (wild pointer), calling `free` twice ("double free"), etc., usually causes a segmentation fault and results in a crash of the program.

These errors can be transient and hard to debug – for example, freed memory is usually not immediately reclaimed by the OS, and thus dangling pointers may persist for a while and appear to work.

On systems where it works, Valgrind is an invaluable tool for identifying what memory is leaked and where it was originally allocated.

Creating pointers to stack variables

Creating a pointer does not extend the life of the variable being pointed to. For example:

```
int* myFunction()
{
    int x = 10;
    return &x;
}
```

Here, `x` has *automatic storage duration* (commonly known as *stack* allocation). Because it is allocated on the stack, its lifetime is only as long as `myFunction` is executing; after `myFunction` has exited, the variable `x` is destroyed. This function gets the address of `x` (using `&x`), and returns it to the caller, leaving the caller with a pointer to a non-existent variable. Attempting to access this variable will then invoke undefined behavior.

Most compilers don't actually clear a stack frame after the function exits, thus dereferencing the returned pointer often gives you the expected data. When another function is called however, the memory being pointed to may be overwritten, and it appears that the data being pointed to has been corrupted.

To resolve this, either `malloc` the storage for the variable to be returned, and return a pointer to the newly created storage, or require that a valid pointer is passed in to the function instead of returning one, for example:

```c
#include <stdlib.h>
#include <stdio.h>

int *solution1(void)
{
    int *x = malloc(sizeof *x);
    if (x == NULL)
    {
        /* Something went wrong */
        return NULL;
    }

    *x = 10;

    return x;
}

void solution2(int *x)
{
    /* NB: calling this function with an invalid or null pointer
       causes undefined behaviour. */

    *x = 10;
}

int main(void)
{
    {
        /* Use solution1() */

        int *foo = solution1();
        if (foo == NULL)
        {
            /* Something went wrong */
            return 1;
        }

        printf("The value set by solution1() is %i\n", *foo);
        /* Will output: "The value set by solution1() is 10" */

        free(foo);    /* Tidy up */
    }

    {
        /* Use solution2() */

        int bar;
        solution2(&bar);

        printf("The value set by solution2() is %i\n", bar);
        /* Will output: "The value set by solution2() is 10" */
    }

    return 0;
```

```
}
```

Incrementing / decrementing and dereferencing

If you write `*p++` to increment what is pointed by `p`, you are wrong.

Post incrementing / decrementing is executed before dereferencing. Therefore, this expression will increment the pointer `p` itself and return what was pointed by `p` before incrementing without changing it.

You should write `(*p)++` to increment what is pointed by `p`.

This rule also applies to post decrementing: `*p--` will decrement the pointer `p` itself, not what is pointed by `p`.

Section 22.3: Dereferencing a Pointer

```
int a = 1;
int *a_pointer = &a;
```

To dereference `a_pointer` and change the value of a, we use the following operation

```
*a_pointer = 2;
```

This can be verified using the following print statements.

```
printf("%d\n", a); /* Prints 2 */
printf("%d\n", *a_pointer); /* Also prints 2 */
```

However, one would be mistaken to dereference a `NULL` or otherwise invalid pointer. This

```
int *p1, *p2;

p1 = (int *) 0xbad;
p2 = NULL;

*p1 = 42;
*p2 = *p1 + 1;
```

is usually undefined behavior. `p1` may not be dereferenced because it points to an address `0xbad` which may not be a valid address. Who knows what's there? It might be operating system memory, or another program's memory. The only time code like this is used, is in embedded development, which stores particular information at hard-coded addresses. `p2` cannot be dereferenced because it is `NULL`, which is invalid.

Section 22.4: Dereferencing a Pointer to a struct

Let's say we have the following structure:

```
struct MY_STRUCT
{
    int my_int;
    float my_float;
};
```

We can define `MY_STRUCT` to omit the `struct` keyword so we don't have to type `struct MY_STRUCT` each time we use it. This, however, is optional.

```
typedef struct MY_STRUCT MY_STRUCT;
```

If we then have a pointer to an instance of this struct

```
MY_STRUCT *instance;
```

If this statement appears at file scope, `instance` will be initialized with a null pointer when the program starts. If this statement appears inside a function, its value is undefined. The variable must be initialized to point to a valid `MY_STRUCT` variable, or to dynamically allocated space, before it can be dereferenced. For example:

```
MY_STRUCT info = { 1, 3.141593F };
MY_STRUCT *instance = &info;
```

When the pointer is valid, we can dereference it to access its members using one of two different notations:

```
int a = (*instance).my_int;
float b = instance->my_float;
```

While both these methods work, it is better practice to use the arrow `->` operator rather than the combination of parentheses, the dereference `*` operator and the dot `.` operator because it is easier to read and understand, especially with nested uses.

Another important difference is shown below:

```
MY_STRUCT copy = *instance;
copy.my_int    = 2;
```

In this case, `copy` contains a copy of the contents of `instance`. Changing `my_int` of `copy` will not change it in `instance`.

```
MY_STRUCT *ref = instance;
ref->my_int    = 2;
```

In this case, `ref` is a reference to `instance`. Changing `my_int` using the reference will change it in `instance`.

It is common practice to use pointers to structs as parameters in functions, rather than the structs themselves. Using the structs as function parameters could cause the stack to overflow if the struct is large. Using a pointer to a struct only uses enough stack space for the pointer, but can cause side effects if the function changes the struct which is passed into the function.

Section 22.5: Const Pointers

Single Pointers

- Pointer to an `int`

  The pointer can point to different integers and the `int`'s can be changed through the pointer. This sample of code assigns b to point to `int b` then changes `b`'s value to `100`.

  ```
  int b;
  int* p;
  p = &b;    /* OK */
  *p = 100;  /* OK */
  ```

- Pointer to a `const int`

  The pointer can point to different integers but the `int`'s value can't be changed through the pointer.

```
int b;
const int* p;
p = &b;    /* OK */
*p = 100;  /* Compiler Error */
```

- `const` pointer to `int`

  The pointer can only point to one `int` but the `int`'s value can be changed through the pointer.

```
int a, b;
int* const p = &b; /* OK as initialisation, no assignment */
*p = 100;  /* OK */
p = &a;    /* Compiler Error */
```

- `const` pointer to `const int`

  The pointer can only point to one `int` and the `int` can not be changed through the pointer.

```
int a, b;
const int* const p = &b; /* OK as initialisation, no assignment */
p = &a;   /* Compiler Error */
*p = 100; /* Compiler Error */
```

Pointer to Pointer

- Pointer to a pointer to an `int`

  This code assigns the address of `p1` to the to double pointer `p` (which then points to `int* p1` (which points to `int`)).

  Then changes `p1` to point to `int a`. Then changes the value of a to be 100.

```
void f1(void)
{
  int a, b;
  int *p1;
  int **p;
  p1 = &b; /* OK */
  p = &p1; /* OK */
  *p = &a; /* OK */
  **p = 100; /* OK */
}
```

- Pointer to pointer to a `const int`

```
 void f2(void)
{
  int b;
  const int *p1;
  const int **p;
```

```
  p = &p1; /* OK */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location ‘**p’ */
}
```

- Pointer to `const` pointer to an `int`

```
void f3(void)
{
  int b;
  int *p1;
  int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location ‘*p’ */
  **p = 100; /* OK */
}
```

- `const` pointer to pointer to `int`

```
void f4(void)
{
  int b;
  int *p1;
  int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable ‘p’ */
  *p = &b; /* OK */
  **p = 100; /* OK */
}
```

- Pointer to `const` pointer to `const int`

```
void f5(void)
{
  int b;
  const int *p1;
  const int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location ‘*p’ */
  **p = 100; /* error: assignment of read-only location ‘**p’ */
}
```

- `const` pointer to pointer to `const int`

```
void f6(void)
{
  int b;
  const int *p1;
  const int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable ‘p’ */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location ‘**p’ */
}
```

- `const` pointer to `const` pointer to `int`

```
void f7(void)
{
```

```
    int b;
    int *p1;
    int * const * const p = &p1; /* OK as initialisation, not assignment */
    p = &p1; /* error: assignment of read-only variable ‘p’  */
    *p = &b; /* error: assignment of read-only location ‘*p’ */
    **p = 100; /* OK */
}
```

Section 22.6: Function pointers

Pointers can also be used to point at functions.

Let's take a basic function:

```
int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}
```

Now, let's define a pointer of that function's type:

```
int (*my_pointer)(int, int);
```

To create one, just use this template:

```
return_type_of_func (*my_func_pointer)(type_arg1, type_arg2, ...)
```

We then must assign this pointer to the function:

```
my_pointer = &my_function;
```

This pointer can now be used to call the function:

```
/* Calling the pointed function */
int result = (*my_pointer)(4, 2);

...

/* Using the function pointer as an argument to another function */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);

    printf("%d\n", result);
}
```

Although this syntax seems more natural and coherent with basic types, attributing and dereferencing function pointers don't require the usage of `&` and `*` operators. So the following snippet is equally valid:

```
/* Attribution without the & operator */
my_pointer = my_function;

/* Dereferencing without the * operator */
int result = my_pointer(4, 2);
```

To increase the readability of function pointers, typedefs may be used.

```
typedef void (*Callback)(int a);

void some_function(Callback callback)
{
    int a = 4;
    callback(a);
}
```

Another readability trick is that the C standard allows one to simplify a function pointer in arguments like above (but not in variable declaration) to something that looks like a function prototype; thus the following can be equivalently used for function definitions and declarations:

```
void some_function(void callback(int))
{
    int a = 4;
    callback(a);
}
```

See also

Function Pointers

Section 22.7: Polymorphic behaviour with void pointers

The `qsort()` standard library function is a good example of how one can use void pointers to make a single function operate on a large variety of different types.

```
void qsort (
    void *base,                                 /* Array to be sorted */
    size_t num,                                 /* Number of elements in array */
    size_t size,                                /* Size in bytes of each element */
    int (*compar)(const void *, const void *)); /* Comparison function for two elements */
```

The array to be sorted is passed as a void pointer, so an array of any type of element can be operated on. The next two arguments tell `qsort()` how many elements it should expect in the array, and how large, in bytes, each element is.

The last argument is a function pointer to a comparison function which itself takes two void pointers. By making the caller provide this function, `qsort()` can effectively sort elements of any type.

Here's an example of such a comparison function, for comparing floats. Note that any comparison function passed to `qsort()` needs to have this type signature. The way it is made polymorphic is by casting the void pointer arguments to pointers of the type of element we wish to compare.

```
int compare_floats(const void *a, const void *b)
{
    float fa = *((float *)a);
    float fb = *((float *)b);
    if (fa < fb)
        return -1;
    if (fa > fb)
        return 1;
    return 0;
}
```

Since we know that qsort will use this function to compare floats, we cast the void pointer arguments back to float pointers before dereferencing them.

Now, the usage of the polymorphic function qsort on an array "array" with length "len" is very simple:

```
qsort(array, len, sizeof(array[0]), compare_floats);
```

Section 22.8: Address-of Operator (&)

For any object (i.e, variable, array, union, struct, pointer or function) the unary address operator can be used to access the address of that object.

Suppose that

```
int i = 1;
int *p = NULL;
```

So then a statement `p = &i;`, copies the address of the variable `i` to the pointer `p`.

It's expressed as `p` ***points to*** `i`.

`printf("%d\n", *p);` prints 1, which is the value of `i`.

Section 22.9: Initializing Pointers

Pointer initialization is a good way to avoid wild pointers. The initialization is simple and is no different from initialization of a variable.

```
#include <stddef.h>

int main()
{
    int *p1 = NULL;
    char *p2 = NULL;
    float *p3 = NULL;

         /* NULL is a macro defined in stddef.h, stdio.h, stdlib.h, and string.h */

    ...
}
```

In most operating systems, inadvertently using a pointer that has been initialized to `NULL` will often result in the program crashing immediately, making it easy to identify the cause of the problem. Using an uninitialized pointer can often cause hard-to-diagnose bugs.

Caution:

The result of dereferencing a `NULL` pointer is undefined, so it *will not necessarily cause a crash* even if that is the natural behaviour of the operating system the program is running on. Compiler optimizations may mask the crash, cause the crash to occur before or after the point in the source code at which the null pointer dereference occurred, or cause parts of the code that contains the null pointer dereference to be unexpectedly removed from the program. Debug builds will not usually exhibit these behaviours, but this is not guaranteed by the language standard. Other unexpected and/or undesirable behaviour is also allowed.

Because `NULL` never points to a variable, to allocated memory, or to a function, it is safe to use as a guard value.

Caution:

Usually `NULL` is defined as `(void *)0`. But this does not imply that the assigned memory address is `0x0`. For more clarification refer to [C-faq for NULL pointers]

Note that you can also initialize pointers to contain values other than NULL.

```
int i1;

int main()
{
   int *p1 = &i1;
   const char *p2 = "A constant string to point to";
   float *p3 = malloc(10 * sizeof(float));
}
```

Section 22.10: Pointer to Pointer

In C, a pointer can refer to another pointer.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &pA;
  int*** pppA = &ppA;

  printf("%d", ***pppA); /* prints 42 */

  return EXIT_SUCCESS;
}
```

But, reference-and-reference directly is not allowed.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &&A; /* Compilation error here! */
  int*** pppA = &&&A;  /* Compilation error here! */

  ...
```

Section 22.11: void* pointers as arguments and return values to standard functions

Version > K&R

`void*` is a catch all type for pointers to object types. An example of this in use is with the `malloc` function, which is declared as

```
void* malloc(size_t);
```

The pointer-to-void return type means that it is possible to assign the return value from `malloc` to a pointer to any other type of object:

```
int* vector = malloc(10 * sizeof *vector);
```

It is generally considered good practice to *not* explicitly cast the values into and out of void pointers. In specific case of `malloc()` this is because with an explicit cast, the compiler may otherwise assume, but not warn about, an incorrect return type for `malloc()`, if you forget to include `stdlib.h`. It is also a case of using the correct behavior of void pointers to better conform to the DRY (don't repeat yourself) principle; compare the above to the following, wherein the following code contains several needless additional places where a typo could cause issues:

```
int* vector = (int*)malloc(10 * sizeof int*);
```

Similarly, functions such as

```
void* memcpy(void *restrict target, void const *restrict source, size_t size);
```

have their arguments specified as `void *` because the address of any object, regardless of the type, can be passed in. Here also, a call should not use a cast

```
unsigned char buffer[sizeof(int)];
int b = 67;
memcpy(buffer, &b, sizeof buffer);
```

Section 22.12: Same Asterisk, Different Meanings

Premise

The most confusing thing surrounding pointer syntax in C and C++ is that there are actually two different meanings that apply when the pointer symbol, the asterisk (`*`), is used with a variable.

Example

Firstly, you use `*` to **declare** a pointer variable.

```
int i = 5;
/* 'p' is a pointer to an integer, initialized as NULL */
int *p = NULL;
/* '&i' evaluates into address of 'i', which then assigned to 'p' */
p = &i;
/* 'p' is now holding the address of 'i' */
```

When you're not declaring (or multiplying), `*` is used to **dereference** a pointer variable:

```
*p = 123;
/* 'p' was pointing to 'i', so this changes value of 'i' to 123 */
```

When you want an existing pointer variable to hold address of other variable, you **don't** use `*`, but do it like this:

```
p = &another_variable;
```

A common confusion among C-programming newbies arises when they declare and initialize a pointer variable at the same time.

```c
int *p = &i;
```

Since `int i = 5;` and `int i; i = 5;` give the same result, some of them might thought `int *p = &i;` and `int *p; *p = &i;` give the same result too. The fact is, no, `int *p; *p = &i;` will attempt to deference an **uninitialized** pointer which will result in UB. Never use `*` when you're not declaring nor dereferencing a pointer.

Conclusion

The asterisk (`*`) has two distinct meanings within C in relation to pointers, depending on where it's used. When used within a **variable declaration**, the value on the right hand side of the equals side should be a ***pointer value*** to an ***address*** in memory. When used with an already **declared variable**, the asterisk will ***dereference*** the pointer value, following it to the pointed-to place in memory, and allowing the value stored there to be assigned or retrieved.

Takeaway

It is important to mind your P's and Q's, so to speak, when dealing with pointers. Be mindful of when you're using the asterisk, and what it means when you use it there. Overlooking this tiny detail could result in buggy and/or undefined behavior that you really don't want to have to deal with.

Chapter 23: Sequence points

Section 23.1: Unsequenced expressions

Version ≥ C11

The following expressions are *unsequenced*:

```
a + b;
a - b;
a * b;
a / b;
a % b;
a & b;
a | b;
```

In the above examples, the expression `a` may be evaluated before or after the expression `b`, `b` may be evaluated before `a`, or they may even be intermixed if they correspond to several instructions.

A similar rule holds for function calls:

```
f(a, b);
```

Here not only `a` and `b` are unsequenced (i.e. the `,` operator in a function call *does not* produce a sequence point) but also `f`, the expression that determines the function that is to be called.

Side effects may be applied immediately after evaluation or deferred until a later point.

Expressions like

```
x++ & x++;
f(x++, x++); /* the ',' in a function call is *not* the same as the comma operator */
x++ * x++;
a[i] = i++;
```

or

```
x++ & x;
f(x++, x);
x++ * x;
a[i++] = i;
```

will yield *undefined behavior* because

- a modification of an object and any other access to it must be sequenced
- the order of evaluation and the order in which *side effects*1 are applied is not specified.

1 Any changes in the state of the execution environment.

Section 23.2: Sequenced expressions

The following expressions are *sequenced*:

```
a && b
a || b
```

```
a , b
a ? b : c
for ( a ; b ; c ) { ... }
```

In all cases, the expression `a` is fully evaluated and *all side effects are applied* before either `b` or `c` are evaluated. In the fourth case, only one of `b` or `c` will be evaluated. In the last case, `b` is fully evaluated and all side effects are applied before `c` is evaluated.

In all cases, the evaluation of expression `a` is *sequenced before* the evaluations of `b` or `c` (alternately, the evaluations of `b` and `c` are *sequenced after* the evaluation of `a`).

Thus, expressions like

```
x++ && x++
x++ ? x++ : y++
(x = f()) && x != 0
for ( x = 0; x < 10; x++ ) { ... }
y = (x++, x++);
```

have well defined behavior.

Section 23.3: Indeterminately sequenced expressions

Function calls as `f(a)` always imply a sequence point between the evaluation of the arguments and the designator (here `f` and `a`) and the actual call. If two such calls are unsequenced, the two function calls are indeterminately sequenced, that is, one is executed before the other, and order is unspecified.

```
unsigned counter = 0;

unsingned account(void) {
   return counter++;
}

int main(void) {
   printf("the order is %u %u\n", account(), account());
}
```

This implicit twofold modification of `counter` during the evaluation of the `printf` arguments is valid, we just don't know which of the calls comes first. As the order is unspecified, it may vary and cannot be depended on. So the printout could be:

> the order is 0 1

or

> the order is 1 0

The analogous statement to the above without intermediate function call

```
   printf("the order is %u %u\n", counter++, counter++); // undefined behavior
```

has undefined behavior because there is no sequence point between the two modifications of `counter`.

Chapter 24: Function Pointers

Function pointers are pointers that point to functions instead of data types. They can be used to allow variability in the function that is to be called, at run-time.

Section 24.1: Introduction

Just like `char` and `int`, a function is a fundamental feature of C. As such, you can declare a pointer to one: which means that you can pass *which function to call* to another function to help it do its job. For example, if you had a `graph()` function that displayed a graph, you could pass *which function to graph* into `graph()`.

```
// A couple of external definitions to make the example clearer
extern unsigned int screenWidth;
extern void plotXY(double x, double y);

// The graph() function.
// Pass in the bounds: the minimum and maximum X and Y that should be plotted.
// Also pass in the actual function to plot.
void graph(double minX, double minY,
           double maxX, double maxY,
           ???? *fn) {            // See below for syntax

    double stepX = (maxX - minX) / screenWidth;
    for (double x=minX; x<maxX; x+=stepX) {

        double y = fn(x);         // Get y for this x by calling passed-in fn()

        if (minY<=y && y<maxY) {
            plotXY(x, y);         // Plot calculated point
        } // if
    } for
} // graph(minX, minY, maxX, maxY, fn)
```

Usage

So the above code will graph whatever function you passed into it - as long as that function meets certain criteria: namely, that you pass a `double` in and get a `double` out. There are many functions like that - `sin()`, `cos()`, `tan()`, `exp()` etc. - but there are many that aren't, such as `graph()` itself!

Syntax

So how do you specify which functions you can pass into `graph()` and which ones you can't? The conventional way is by using a syntax that may not be easy to read or understand:

```
double (*fn)(double); // fn is a pointer-to-function that takes a double and returns one
```

The problem above is that there are two things trying to be defined at the same time: the structure of the function, and the fact that it's a pointer. So, split the two definitions! But by using `typedef`, a better syntax (easier to read & understand) can be achieved.

Section 24.2: Returning Function Pointers from a Function

```
#include <stdio.h>

enum Op
{
```

```
  ADD = '+',
  SUB = '-',
};


/* add: add a and b, return result */
int add(int a, int b)
{
    return a + b;
}

/* sub: subtract b from a, return result */
int sub(int a, int b)
{
    return a - b;
}

/* getmath: return the appropriate math function */
int (*getmath(enum Op op))(int,int)
{
    switch (op)
    {
        case ADD:
            return &add;
        case SUB:
            return &sub;
        default:
            return NULL;
    }
}

int main(void)
{
    int a, b, c;
    int (*fp)(int,int);

    fp = getmath(ADD);

    a = 1, b = 2;
    c = (*fp)(a, b);
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

Section 24.3: Best Practices

Using typedef

It might be handy to use a `typedef` instead of declaring the function pointer each time by hand.

The syntax for declaring a `typedef` for a function pointer is:

```
typedef returnType (*name)(parameters);
```

Example:

Posit that we have a function, `sort`, that expects a function pointer to a function `compare` such that:

> compare - A compare function for two elements which is to be supplied to a sort function.

> "compare" is expected to return 0 if the two elements are deemed equal, a positive value if the first element passed is "larger" in some sense than the latter element and otherwise the function returns a negative value (meaning that the first element is "lesser" than the latter).

Without a `typedef` we would pass a function pointer as an argument to a function in the following manner:

```
void sort(int (*compare)(const void *elem1, const void *elem2)) {
    /* inside of this block, the function is named "compare" */
}
```

With a `typedef`, we'd write:

```
typedef int (*compare_func)(const void *, const void *);
```

and then we could change the function signature of `sort` to:

```
void sort(compare_func func) {
    /* In this block the function is named "func" */
}
```

both definitions of `sort` would accept any function of the form

```
int compare(const void *arg1, const void *arg2) {
    /* Note that the variable names do not have to be "elem1" and "elem2" */
}
```

Function pointers are the only place where you should include the pointer property of the type, e.g. do not try to define types like `typedef struct something_struct *something_type`. This applies even for a structure with members which are not supposed to accessed directly by API callers, for example the stdio.h `FILE` type (which as you now will notice is not a pointer).

Taking context pointers.

A function pointer should almost always take a user-supplied void * as a context pointer.

Example

```
/* function minimiser, details unimportant */
double findminimum( double (*fptr)(double x, double y, void *ctx), void *ctx)
{
    ...
    /* repeatedly make calls like this */
    temp = (*fptr)(testx, testy, ctx);
}

/* the function we are minimising, sums two cubics */
double *cubics(double x, double y, void *ctx)
{
    double *coeffsx = ctx;
    double *coeffsy = coeffx + 4;

    return coeffsx[0] * x * x * x + coeffsx[1] * x * x + coeffsx[2] * x + coeffsx[3] +
           coeffsy[0] * y * y * y + coeffsy[1] * y * y + coeffsy[2] * y + coeffsy[3];

}

void caller()
```

```
{
    /* context, the coefficients of the cubics */
    double coeffs[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    double min;

    min = findminimum(cubics, coeffs);
}
```

Using the context pointer means that the extra parameters do not need to be hard-coded into the function pointed to, or require the use globals.

The library function `qsort()` does not follow this rule, and one can often get away without context for trivial comparison functions. But for anything more complicated, the context pointer becomes essential.

See also

Functions pointers

Section 24.4: Assigning a Function Pointer

```
#include <stdio.h>

/* increment: take number, increment it by one, and return it */
int increment(int i)
{
    printf("increment %d by 1\n", i);
    return i + 1;
}

/* decrement: take number, decrement it by one, and return it */
int decrement(int i)
{
    printf("decrement %d by 1\n", i);
    return i - 1;
}

int main(void)
{
    int num = 0;          /* declare number to increment */
    int (*fp)(int);       /* declare a function pointer */

    fp = &increment;      /* set function pointer to increment function */
    num = (*fp)(num);     /* increment num */
    num = (*fp)(num);     /* increment num a second time */

    fp = &decrement;      /* set function pointer to decrement function */
    num = (*fp)(num);     /* decrement num */
    printf("num is now: %d\n", num);
    return 0;
}
```

Section 24.5: Mnemonic for writing function pointers

All C functions are in actuality pointers to a spot in the program memory where some code exists. The main use of a function pointer is to provide a "callback" to other functions (or to simulate classes and objects).

The syntax of a function, as defined further down on this page is:

returnType (*name)(parameters)

A mnemonic for writing a function pointer definition is the following procedure:

1. Begin by writing a normal function declaration: `returnType name(parameters)`
2. Wrap the function name with pointer syntax: `returnType (*name)(parameters)`

Section 24.6: Basics

Just like you can have a pointer to an **int**, **char**, **float**, **array/string**, **struct**, etc. - you can have a pointer to a function.

Declaring the pointer takes the *return value of the function*, the *name of the function*, and the *type of arguments/parameters it receives*.

Say you have the following function declared and initialized:

```
int addInt(int n, int m){
    return n+m;
}
```

You can declare and initialize a pointer to this function:

```
int (*functionPtrAdd)(int, int) = addInt; // or &addInt - the & is optional
```

If you have a void function it could look like this:

```
void Print(void){
    printf("look ma' - no hands, only pointers!\n");
}
```

Then declaring the pointer to it would be:

```
void (*functionPtrPrint)(void) = Print;
```

Accessing the function itself would require dereferencing the pointer:

```
sum = (*functionPtrAdd)(2, 3); //will assign 5 to sum
(*functionPtrPrint)(); //will print the text in Print function
```

As seen in more advanced examples in this document, declaring a pointer to a function could get messy if the function is passed more than a few parameters. If you have a few pointers to functions that have identical "structure" (same type of return value, and same type of parameters) it's best to use the **typedef** command to save you some typing, and to make the code more clear:

```
typedef int (*ptrInt)(int, int);

int Add(int i, int j){
    return i+j;
}

int Multiply(int i, int j){
    return i*j;
}

int main()
```

```
{
    ptrInt ptr1 = Add;
    ptrInt ptr2 = Multiply;

    printf("%d\n", (*ptr1)(2,3)); //will print 5
    printf("%d\n", (*ptr2)(2,3)); //will print 6
    return 0;
}
```

You can also create an **Array of function-pointers**. If all the pointers are of the same "structure":

```
int (*array[2]) (int x, int y); // can hold 2 function pointers
array[0] = Add;
array[1] = Multiply;
```

You can learn more here and here.

It is also possible to define an array of function-pointers of different types, though that would require casting whenever you want to access the specific function. You can learn more here.

Chapter 25: Function Parameters

Section 25.1: Parameters are passed by value

In C, all function parameters are passed by value, so modifying what is passed in callee functions won't affect caller functions' local variables.

```
#include <stdio.h>

void modify(int v) {
    printf("modify 1: %d\n", v); /* 0 is printed */
    v = 42;
    printf("modify 2: %d\n", v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(v);
    printf("main 2: %d\n", v); /* 0 is printed, not 42 */
    return 0;
}
```

You can use pointers to let callee functions modify caller functions' local variables. Note that this is not *pass by reference* but the pointer *values* pointing at the local variables are passed.

```
#include <stdio.h>

void modify(int* v) {
    printf("modify 1: %d\n", *v); /* 0 is printed */
    *v = 42;
    printf("modify 2: %d\n", *v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(&v);
    printf("main 2: %d\n", v); /* 42 is printed */
    return 0;
}
```

However returning the address of a local variable to the callee results in undefined behaviour. See Dereferencing a pointer to variable beyond its lifetime.

Section 25.2: Passing in Arrays to Functions

```
int getListOfFriends(size_t size, int friend_indexes[]) {
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = i;
  }
}
```

Version ≥ C99 Version < C11

```
/* Type "void" and VLAs ("int friend_indexes[static size]") require C99 at least.
   In C11 VLAs are optional. */
void getListOfFriends(size_t size, int friend_indexes[static size]) {
```

```
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = 1;
  }
}
```

Here the `static` inside the `[]` of the function parameter, request that the argument array must have at least as many elements as are specified (i.e. `size` elements). To be able to use that feature we have to ensure that the `size` parameter comes before the array parameter in the list.

Use `getListOfFriends()` like this:

```
#define LIST_SIZE (50)

int main(void) {
  size_t size_of_list = LIST_SIZE;
  int friends_indexes[size_of_list];

  getListOfFriends(size_of_list, friend_indexes); /* friend_indexes decays to a pointer to the
                                                     address of its 1st element:
                                                                      &friend_indexes[0] */

  /* Here friend_indexes carries: {0, 1, 2, ..., 49}; */

  return 0;
}
```

See also

Passing multidimensional arrays to a function

Section 25.3: Order of function parameter execution

The order of execution of parameters is undefined in C programming. Here it may execute from left to right or from right to left. The order depends on the implementation.

```
#include <stdio.h>

void function(int a, int b)
{
    printf("%d %d\n", a, b);
}

int main(void)
{
    int a = 1;
    function(a++, ++a);
    return 0;
}
```

Section 25.4: Using pointer parameters to return multiple values

A common pattern in C, to easily imitate returning multiple values from a function, is to use pointers.

```
#include <stdio.h>
```

```c
void Get( int* c , double* d )
{
    *c = 72;
    *d = 175.0;
}

int main(void)
{
    int a = 0;
    double b = 0.0;

    Get( &a , &b );

    printf("a: %d, b: %f\n", a , b );

    return 0;
}
```

Section 25.5: Example of function returning struct containing values with error codes

Most examples of a function returning a value involve providing a pointer as one of the arguments to allow the function to modify the value pointed to, similar to the following. The actual return value of the function is usually some type such as an `int` to indicate the status of the result, whether it worked or not.

```c
int func (int *pIvalue)
{
    int iRetStatus = 0;             /* Default status is no change */

    if (*pIvalue > 10) {
        *pIvalue = *pIvalue * 45;   /* Modify the value pointed to */
        iRetStatus = 1;             /* indicate value was changed */
    }

    return iRetStatus;              /* Return an error code */
}
```

However you can also use a `struct` as a return value which allows you to return both an error status along with other values as well. For instance.

```c
typedef struct {
    int    iStat;      /* Return status */
    int    iValue;     /* Return value */
}  RetValue;

RetValue func (int iValue)
{
    RetValue iRetStatus = {0, iValue};

    if (iValue > 10) {
        iRetStatus.iValue = iValue * 45;
        iRetStatus.iStat = 1;
    }

    return iRetStatus;
}
```

This function could then be used like the following sample.

```c
int usingFunc (int iValue)
{
    RetValue iRet = func (iValue);

    if (iRet.iStat == 1) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Or it could be used like the following.

```c
int usingFunc (int iValue)
{
    RetValue iRet;

    if ( (iRet = func (iValue)).iStat == 1 ) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Chapter 26: Pass 2D-arrays to functions

Section 26.1: Pass a 2D-array to a function

Passing a 2d array to a functions seems simple and obvious and we happily write:

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int **, int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int **a, int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

But the compiler, here GCC in version 4.9.4 , does not appreciate it well.

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c11 passarr.c -o passarr
passarr.c: In function 'main':
passarr.c:16:8: warning: passing argument 1 of 'fun1' from incompatible pointer type
fun1(array_2D, n, m);
^
passarr.c:8:6: note: expected 'int **' but argument is of type 'int (*)[2]'
void fun1(int **, int, int);
```

The reasons for this are twofold: the main problem is that arrays are not pointers and the second inconvenience is the so called *pointer decay*. Passing an array to a function will decay the array to a pointer to the first element of the array--in the case of a 2d array it decays to a pointer to the first row because in C arrays are sorted row-first.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int, int);
```

```
int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

It *is* necessary to pass the number of rows, they cannot be computed.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int rows = ROWS;

  /* works here because array_2d is still in scope and still an array */
  printf("MAIN: %zu\n",sizeof(array_2D)/sizeof(array_2D[0]));

  fun1(array_2D, rows);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int rows)
{
  int i, j;
  int n, m;

  n = rows;
  /* Works, because that information is passed (as "COLS").
     It is also redundant because that value is known at compile time (in "COLS"). */
  m = (int) (sizeof(a[0])/sizeof(a[0][0]));

  /* Does not work here because the "decay" in "pointer decay" is meant
     literally--information is lost. */
  printf("FUN1: %zu\n",sizeof(a)/sizeof(a[0]));

  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
```

```
    }
  }
}
```
Version = C99

The number of columns is predefined and hence fixed at compile time, but the predecessor to the current C-standard (that was ISO/IEC 9899:1999, current is ISO/IEC 9899:2011) implemented VLAs (TODO: link it) and although the current standard made it optional, almost all modern C-compilers support it (TODO: check if MS Visual Studio supports it now).

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];

  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = (i + 1) * (j + 1);
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
  int i, j;
  int n, m;

  n = rows;
  /* Does not work anymore, no sizes are specified anymore
  m = (int) (sizeof(a[0])/sizeof(a[0][0])); */
  m = cols;


  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

This does not work, the compiler complains:

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function ‘fun1’:
passarr.c:168:7: error: invalid use of array with unspecified bounds
printf("array[%d][%d]=%d\n", i, j, a[i][j]);
```

It becomes a bit clearer if we intentionally make an error in the call of the function by changing the declaration to `void fun1(int **a, int rows, int cols)`. That causes the compiler to complain in a different, but equally nebulous way

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function ‘main’:
passarr.c:208:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type
   fun1(array_2D, rows, cols);
        ^
passarr.c:185:6: note: expected ‘int **’ but argument is of type ‘int (*)[(sizetype)(cols)]’
 void fun1(int **, int rows, int cols);
```

We can react in several ways, one of it is to ignore all of it and do some illegible pointer juggling:

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
  int i, j;
  int n, m;
```

```
  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, *( (*a) + (i * cols + j)));
    }
  }
}
```

Or we do it right and pass the needed information to `fun1`. To do so wee need to rearrange the arguments to `fun1`: the size of the column must come before the declaration of the array. To keep it more readable the variable holding the number of rows has changed its place, too, and is first now.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int (*)[]);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(rows, cols, array_2D);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int (*a)[cols])
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
```

```
  }
}
```

This looks awkward to some people, who hold the opinion that the order of variables should not matter. That is not much of a problem, just declare a pointer and let it point to the array.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int **);

int main(int argc, char **argv)
{
  int rows, cols, i, j;


  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }
  // a "rows" number of pointers to "int". Again a VLA
  int *a[rows];
  // initialize them to point to the individual rows
  for (i = 0; i < rows; i++) {
      a[i] = array_2D[i];
  }

  fun1(rows, cols, a);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int **a)
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Section 26.2: Using flat arrays as 2D arrays

Often the easiest solution is simply to pass 2D and higher arrays around as flat memory.

```
/* create 2D array with dimensions determined at runtime */
double *matrix = malloc(width * height * sizeof(double));

/* initialise it (for the sake of illustration we want 1.0 on the diagonal) */
int x, y;
for (y = 0; y < height; y++)
{
    for (x = 0; x < width; x++)
    {
        if (x == y)
            matrix[y * width + x] = 1.0;
        else
            matrix[y * width + x] = 0.0;
    }
}

/* pass it to a subroutine */
 manipulate_matrix(matrix, width, height);


/* do something with the matrix, e.g. scale by 2 */
void manipulate_matrix(double *matrix, int width, int height)
{
    int x, y;

    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
            matrix[y * width + x] *= 2.0;
        }
    }
}
```

Chapter 27: Error handling

Section 27.1: errno

When a standard library function fails, it often sets errno to the appropriate error code. The C standard requires at least 3 values for errno be set:

Value	Meaning
EDOM	Domain error
ERANGE	Range error
EILSEQ	Illegal multi-byte character sequence

Section 27.2: strerror

If `perror` is not flexible enough, you may obtain a user-readable error description by calling `strerror` from **`<string.h>`**.

```
int main(int argc, char *argv[])
{
    FILE *fout;
    int last_error = 0;

    if ((fout = fopen(argv[1], "w")) == NULL) {
        last_error = errno;
         /* reset errno and continue */
         errno = 0;
    }

    /* do some processing and try opening the file differently, then */


    if (last_error) {
        fprintf(stderr, "fopen: Could not open %s for writing: %s",
                argv[1], strerror(last_error));
        fputs("Cross fingers and continue", stderr);
    }

    /* do some other processing */

    return EXIT_SUCCESS;
}
```

Section 27.3: perror

To print a user-readable error message to `stderr`, call `perror` from <stdio.h>.

```
int main(int argc, char *argv[])
{
   FILE *fout;

   if ((fout = fopen(argv[1], "w")) == NULL) {
      perror("fopen: Could not open file for writing");
      return EXIT_FAILURE;
   }
return EXIT_SUCCESS;
}
```

This will print an error message concerning the current value of `errno`.

Chapter 28: Undefined behavior

In C, some expressions yield *undefined behavior*. The standard explicitly chooses to not define how a compiler should behave if it encounters such an expression. As a result, a compiler is free to do whatever it sees fit and may produce useful results, unexpected results, or even crash.

Code that invokes UB may work as intended on a specific system with a specific compiler, but will likely not work on another system, or with a different compiler, compiler version or compiler settings.

Section 28.1: Dereferencing a pointer to variable beyond its lifetime

```
int* foo(int bar)
{
    int baz = 6;
    baz += bar;
    return &baz; /* (&baz) copied to new memory location outside of foo. */
} /* (1) The lifetime of baz and bar end here as they have automatic storage
   * duration (local variables), thus the returned pointer is not valid! */

int main (void)
{
    int* p;

    p = foo(5);  /* (2) this expression's behavior is undefined */
    *p = *p - 6; /* (3) Undefined behaviour here */

    return 0;
}
```

Some compilers helpfully point this out. For example, `gcc` warns with:

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

and `clang` warns with:

```
warning: address of stack memory associated with local variable 'baz' returned
[-Wreturn-stack-address]
```

for the above code. But compilers may not be able to help in complex code.

(1) Returning reference to variable declared `static` is defined behaviour, as the variable is not destroyed after leaving current scope.

(2) According to ISO/IEC 9899:2011 6.2.4 §2, "The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime."

(3) Dereferencing the pointer returned by the function `foo` is undefined behaviour as the memory it references holds an indeterminate value.

Section 28.2: Copying overlapping memory

A wide variety of standard library functions have among their effects copying byte sequences from one memory

region to another. Most of these functions have undefined behavior when the source and destination regions overlap.

For example, this ...

```
#include <string.h> /* for memcpy() */

char str[19] = "This is an example";
memcpy(str + 7, str, 10);
```

... attempts to copy 10 bytes where the source and destination memory areas overlap by three bytes. To visualize:

```
               overlapping area
                |
               _ _
              |   |
              v   v
T h i s   i s   a n   e x a m p l e \0
^             ^
|             |
|             destination
|
source
```

Because of the overlap, the resulting behavior is undefined.

Among the standard library functions with a limitation of this kind are `memcpy()`, `strcpy()`, `strcat()`, `sprintf()`, and `sscanf()`. The standard says of these and several other functions:

> If copying takes place between objects that overlap, the behavior is undefined.

The `memmove()` function is the principal exception to this rule. Its definition specifies that the function behaves as if the source data were first copied into a temporary buffer and then written to the destination address. There is no exception for overlapping source and destination regions, nor any need for one, so `memmove()` has well-defined behavior in such cases.

The distinction reflects an efficiency *vs*. generality tradeoff. Copying such as these functions perform usually occurs between disjoint regions of memory, and often it is possible to know at development time whether a particular instance of memory copying will be in that category. Assuming non-overlap affords comparatively more efficient implementations that do not reliably produce correct results when the assumption does not hold. Most C library functions are allowed the more efficient implementations, and `memmove()` fills in the gaps, serving the cases where the source and destination may or do overlap. To produce the correct effect in all cases, however, it must perform additional tests and / or employ a comparatively less efficient implementation.

Section 28.3: Signed integer overflow

Per paragraph 6.5/5 of both C99 and C11, evaluation of an expression produces undefined behavior if the result is not a representable value of the expression's type. For arithmetic types, that's called an *overflow*. Unsigned integer arithmetic does not overflow because paragraph 6.2.5/9 applies, causing any unsigned result that otherwise would be out of range to be reduced to an in-range value. There is no analogous provision for *signed* integer types, however; these can and do overflow, producing undefined behavior. For example,

```
#include <limits.h>      /* to get INT_MAX */
```

```
int main(void) {
    int i = INT_MAX + 1; /* Overflow happens here */
    return 0;
}
```

Most instances of this type of undefined behavior are more difficult to recognize or predict. Overflow can in principle arise from any addition, subtraction, or multiplication operation on signed integers (subject to the usual arithmetic conversions) where there are not effective bounds on or a relationship between the operands to prevent it. For example, this function:

```
int square(int x) {
    return x * x;  /* overflows for some values of x */
}
```

is reasonable, and it does the right thing for small enough argument values, but its behavior is undefined for larger argument values. You cannot judge from the function alone whether programs that call it exhibit undefined behavior as a result. It depends on what arguments they pass to it.

On the other hand, consider this trivial example of overflow-safe signed integer arithmetic:

```
int zero(int x) {
    return x - x;  /* Cannot overflow */
}
```

The relationship between the operands of the subtraction operator ensures that the subtraction never overflows. Or consider this somewhat more practical example:

```
int sizeDelta(FILE *f1, FILE *f2) {
    int count1 = 0;
    int count2 = 0;
    while (fgetc(f1) != EOF) count1++;  /* might overflow */
    while (fgetc(f2) != EOF) count2++;  /* might overflow */

    return count1 - count2; /* provided no UB to this point, will not overflow */
}
```

As long as that the counters do not overflow individually, the operands of the final subtraction will both be non-negative. All differences between any two such values are representable as `int`.

Section 28.4: Use of an uninitialized variable

```
int a;
printf("%d", a);
```

The variable a is an `int` with automatic storage duration. The example code above is trying to print the value of an uninitialized variable (a was never initialized). Automatic variables which are not initialized have indeterminate values; accessing these can lead to undefined behavior.

Note: Variables with static or thread local storage, including global variables without the `static` keyword, are initialized to either zero, or their initialized value. Hence the following is legal.

```
static int b;
printf("%d", b);
```

A very common mistake is to not initialize the variables that serve as counters to 0. You add values to them, but

since the initial value is garbage, you will invoke **Undefined Behavior**, such as in the question Compilation on terminal gives off pointer warning and strange symbols.

Example:

```
#include <stdio.h>

int main(void) {
    int i, counter;
    for(i = 0; i < 10; ++i)
        counter += i;
    printf("%d\n", counter);
    return 0;
}
```

Output:

```
C02QT2UBFVH6-lm:~ gsamaras$ gcc main.c -Wall -o main
main.c:6:9: warning: variable 'counter' is uninitialized when used here [-Wuninitialized]
        counter += i;
        ^~~~~~~
main.c:4:19: note: initialize the variable 'counter' to silence this warning
    int i, counter;
                  ^
                   = 0
1 warning generated.
C02QT2UBFVH6-lm:~ gsamaras$ ./main
32812
```

The above rules are applicable for pointers as well. For example, the following results in undefined behavior

```
int main(void)
{
    int *p;
    p++; // Trying to increment an uninitialized pointer.
}
```

Note that the above code on its own might not cause an error or segmentation fault, but trying to dereference this pointer later would cause the undefined behavior.

Section 28.5: Data race

Version ≥ C11

C11 introduced support for multiple threads of execution, which affords the possibility of data races. A program contains a data race if an object in it is accessed1 by two different threads, where at least one of the accesses is non-atomic, at least one modifies the object, and program semantics fail to ensure that the two accesses cannot overlap temporally.2 Note well that actual concurrency of the accesses involved is not a condition for a data race; data races cover a broader class of issues arising from (allowed) inconsistencies in different threads' views of memory.

Consider this example:

```
#include <threads.h>

int a = 0;
```

```
int Function( void* ignore )
{
    a = 1;

    return 0;
}

int main( void )
{
    thrd_t id;
    thrd_create( &id , Function , NULL );

    int b = a;

    thrd_join( id , NULL );
}
```

The main thread calls `thrd_create` to start a new thread running function `Function`. The second thread modifies `a`, and the main thread reads `a`. Neither of those access is atomic, and the two threads do nothing either individually or jointly to ensure that they do not overlap, so there is a data race.

Among the ways this program could avoid the data race are

- the main thread could perform its read of `a` before starting the other thread;
- the main thread could perform its read of `a` after ensuring via `thrd_join` that the other has terminated;
- the threads could synchronize their accesses via a mutex, each one locking that mutex before accessing `a` and unlocking it afterward.

As the mutex option demonstrates, avoiding a data race does not require ensuring a specific order of operations, such as the child thread modifying `a` before the main thread reads it; it is sufficient (for avoiding a data race) to ensure that for a given execution, one access will happen before the other.

1 Modifying or reading an object.

2 (Quoted from ISO:IEC 9889:201x, section 5.1.2.4 "Multi-threaded executions and data races")
The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

Section 28.6: Read value of pointer that was freed

Even just **reading** the value of a pointer that was freed (i.e. without trying to dereference the pointer) is undefined behavior(UB), e.g.

```
char *p = malloc(5);
free(p);
if (p == NULL) /* NOTE: even without dereferencing, this may have UB */
{

}
```

Quoting **ISO/IEC 9899:2011**, section 6.2.4 §2:

> […] The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

The use of indeterminate memory for anything, including apparently harmless comparison or arithmetic, can have undefined behavior if the value can be a trap representation for the type.

Section 28.7: Using incorrect format specifier in printf

Using an incorrect format specifier in the first argument to `printf` invokes undefined behavior. For example, the code below invokes undefined behavior:

```
long z = 'B';
printf("%c\n", z);
```

Here is another example

```
printf("%f\n",0);
```

Above line of code is undefined behavior. `%f` expects double. However 0 is of type `int`.

Note that your compiler usually can help you avoid cases like these, if you turn on the proper flags during compiling (`-Wformat` in `clang` and `gcc`). From the last example:

```
warning: format specifies type 'double' but the argument has type
      'int' [-Wformat]
    printf("%f\n",0);
            ~~    ^
            %d
```

Section 28.8: Modify string literal

In this code example, the char pointer `p` is initialized to the address of a string literal. Attempting to modify the string literal has undefined behavior.

```
char *p = "hello world";
p[0] = 'H'; // Undefined behavior
```

However, modifying a mutable array of `char` directly, or through a pointer is naturally not undefined behavior, even if its initializer is a literal string. The following is fine:

```
char a[] = "hello, world";
char *p = a;

a[0] = 'H';
p[7] = 'W';
```

That's because the string literal is effectively copied to the array each time the array is initialized (once for variables with static duration, each time the array is created for variables with automatic or thread duration — variables with allocated duration aren't initialized), and it is fine to modify array contents.

Section 28.9: Passing a null pointer to printf %s conversion

The `%s` conversion of `printf` states that the corresponding argument *a pointer to the initial element of an array of character type*. A null pointer does not point to the initial element of any array of character type, and thus the behavior of the following is undefined:

```
char *foo = NULL;
```

```c
printf("%s", foo); /* undefined behavior */
```

However, the undefined behavior does not always mean that the program crashes — some systems take steps to avoid the crash that normally happens when a null pointer is dereferenced. For example Glibc is known to print

```
(null)
```

for the code above. However, add (just) a newline to the format string and you will get a crash:

```c
char *foo = 0;
printf("%s\n", foo); /* undefined behavior */
```

In this case, it happens because GCC has an optimization that turns `printf("%s\n", argument);` into a call to `puts` with `puts(argument)`, and `puts` in Glibc does not handle null pointers. All this behavior is standard conforming.

Note that *null pointer* is different from an *empty string*. So, the following is valid and has no undefined behaviour. It'll just print a *newline*:

```c
char *foo = "";
printf("%s\n", foo);
```

Section 28.10: Modifying any object more than once between two sequence points

```c
int i = 42;
i = i++; /* Assignment changes variable, post-increment as well */
int a = i++ + i--;
```

Code like this often leads to speculations about the "resulting value" of `i`. Rather than specifying an outcome, however, the C standards specify that evaluating such an expression produces *undefined behavior*. Prior to C2011, the standard formalized these rules in terms of so-called *sequence points*:

> Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.

(C99 standard, section 6.5, paragraph 2)

That scheme proved to be a little too coarse, resulting in some expressions exhibiting undefined behavior with respect to C99 that plausibly should not do. C2011 retains sequence points, but introduces a more nuanced approach to this area based on *sequencing* and a relationship it calls "sequenced before":

> If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.

(C2011 standard, section 6.5, paragraph 2)

The full details of the "sequenced before" relation are too long to describe here, but they supplement sequence points rather than supplanting them, so they have the effect of defining behavior for some evaluations whose

behavior previously was undefined. In particular, if there is a sequence point between two evaluations, then the one before the sequence point is "sequenced before" the one after.

The following example has well-defined behaviour:

```
int i = 42;
i = (i++, i+42); /* The comma-operator creates a sequence point */
```

The following example has undefined behaviour:

```
int i = 42;
printf("%d %d\n", i++, i++); /* commas as separator of function arguments are not comma-operators */
```

As with any form of undefined behavior, observing the actual behavior of evaluating expressions that violate the sequencing rules is not informative, except in a retrospective sense. The language standard provides no basis for expecting such observations to be predictive even of the future behavior of the same program.

Section 28.11: Freeing memory twice

Freeing memory twice is undefined behavior, e.g.

```
int * x = malloc(sizeof(int));
*x = 9;
free(x);
free(x);
```

Quote from standard(7.20.3.2. The free function of C99):

> Otherwise, if the argument does not match a pointer earlier returned by the calloc, malloc, or realloc function, or if the space has been deallocated by a call to free or realloc, the behavior is undefined.

Section 28.12: Bit shifting using negative counts or beyond the width of the type

If the *shift count* value is a **negative value** then both *left shift* and *right shift* operations are undefined1:

```
int x = 5 << -3; /* undefined */
int x = 5 >> -3; /* undefined */
```

If *left shift* is performed on a **negative value**, it's undefined:

```
int x = -5 << 3; /* undefined */
```

If *left shift* is performed on a **positive value** and result of the mathematical value is **not** representable in the type, it's undefined1:

```
/* Assuming an int is 32-bits wide, the value '5 * 2^72' doesn't fit
 * in an int. So, this is undefined. */

int x = 5 << 72;
```

Note that *right shift* on a **negative value** (.e.g `-5 >> 3`) is *not* undefined but *implementation-defined*.

1 Quoting *ISO/IEC 9899:201x*, section 6.5.7:

> If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

Section 28.13: Returning from a function that's declared with `_Noreturn` or `noreturn` function specifier

Version ≥ C11

The function specifier `_Noreturn` was introduced in C11. The header **`<stdnoreturn.h>`** provides a macro `noreturn` which expands to `_Noreturn`. So using `_Noreturn` or `noreturn` from **`<stdnoreturn.h>`** is fine and equivalent.

A function that's declared with `_Noreturn` (or `noreturn`) is not allowed to return to its caller. If such a function *does* return to its caller, the behavior is undefined.

In the following example, `func()` is declared with `noreturn` specifier but it returns to its caller.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void func(void);

void func(void)
{
    printf("In func()...\n");
} /* Undefined behavior as func() returns */

int main(void)
{
    func();
    return 0;
}
```

`gcc` and `clang` produce warnings for the above program:

```
$ gcc test.c
test.c: In function ‘func’:
test.c:9:1: warning: ‘noreturn’ function does return
}
^
$ clang test.c
test.c:9:1: warning: function declared 'noreturn' should not return [-Winvalid-noreturn]
}
^
```

An example using `noreturn` that has well-defined behavior:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void my_exit(void);
```

```
/* calls exit() and doesn't return to its caller. */
void my_exit(void)
{
    printf("Exiting...\n");
    exit(0);
}

int main(void)
{
    my_exit();
    return 0;
}
```

Section 28.14: Accessing memory beyond allocated chunk

A a pointer to a piece of memory containing `n` elements may only be dereferenced if it is in the range `memory` and `memory + (n - 1)`. Dereferencing a pointer outside of that range results in undefined behavior. As an example, consider the following code:

```
int array[3];
int *beyond_array = array + 3;
*beyond_array = 0; /* Accesses memory that has not been allocated. */
```

The third line accesses the 4th element in an array that is only 3 elements long, leading to undefined behavior. Similarly, the behavior of the second line in the following code fragment is also not well defined:

```
int array[3];
array[3] = 0;
```

Note that pointing past the last element of an array is not undefined behavior (`beyond_array = array + 3` is well defined here), but dereferencing it is (`*beyond_array` is undefined behavior). This rule also holds for dynamically allocated memory (such as buffers created through `malloc`).

Section 28.15: Modifying a const variable using a pointer

```
int main (void)
{
    const int foo_readonly = 10;
    int *foo_ptr;

    foo_ptr = (int *)&foo_readonly; /* (1) This casts away the const qualifier */
    *foo_ptr = 20; /* This is undefined behavior */

    return 0;
}
```

Quoting *ISO/IEC 9899:201x*, section 6.7.3 §2:

> If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. [...]

(1) In GCC this can throw the following warning: `warning: assignment discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]`

Section 28.16: Reading an uninitialized object that is not backed by memory

Version ≥ C11

Reading an object will cause undefined behavior, if the object is1:

- uninitialized
- defined with automatic storage duration
- it's address is never taken

The variable a in the below example satisfies all those conditions:

```
void Function( void )
{
    int a;
    int b = a;
}
```

1 (Quoted from: ISO:IEC 9899:201X 6.3.2.1 Lvalues, arrays, and function designators 2)
If the lvalue designates an object of automatic storage duration that could have been declared with the register storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

Section 28.17: Addition or subtraction of pointer not properly bounded

The following code has undefined behavior:

```
char buffer[6] = "hello";
char *ptr1 = buffer - 1;  /* undefined behavior */
char *ptr2 = buffer + 5;  /* OK, pointing to the '\0' inside the array */
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char *ptr4 = buffer + 7;  /* undefined behavior */
```

According to C11, if addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object, the behavior is undefined (6.5.6).

Additionally it is naturally undefined behavior to *dereference* a pointer that points to just beyond the array:

```
char buffer[6] = "hello";
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char value = *ptr3;       /* undefined behavior */
```

Section 28.18: Dereferencing a null pointer

This is an example of dereferencing a NULL pointer, causing undefined behavior.

```
int * pointer = NULL;
int value = *pointer; /* Dereferencing happens here */
```

A `NULL` pointer is guaranteed by the C standard to compare unequal to any pointer to a valid object, and dereferencing it invokes undefined behavior.

Section 28.19: Using fflush on an input stream

The POSIX and C standards explicitly state that using `fflush` on an input stream is undefined behavior. The `fflush` is defined only for output streams.

```
#include <stdio.h>

int main()
{
    int i;
    char input[4096];

    scanf("%i", &i);
    fflush(stdin); // <-- undefined behavior
    gets(input);

    return 0;
}
```

There is no standard way to discard unread characters from an input stream. On the other hand, some implementations uses `fflush` to clear `stdin` buffer. Microsoft defines the behavior of `fflush` on an input stream: If the stream is open for input, `fflush` clears the contents of the buffer. According to POSIX.1-2008, the behavior of `fflush` is undefined unless the input file is seekable.

See Using `fflush(stdin)` for many more details.

Section 28.20: Inconsistent linkage of identifiers

```
extern int var;
static int var; /* Undefined behaviour */
```

C11, §6.2.2, 7 says:

> If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

Note that if an prior declaration of an identifier is visible then it'll have the prior declaration's linkage. *C11, §6.2.2, 4* allows it:

> For an identifier declared with the storage-class specifier extern in a scope in which a prior declaration of that identifier is visible,31) if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

```
/* 1. This is NOT undefined */
static int var;
extern int var;


/* 2. This is NOT undefined */
static int var;
static int var;
```

```
/* 3. This is NOT undefined */
extern int var;
extern int var;
```

Section 28.21: Missing return statement in value returning function

```
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* Trying to use the (not) returned value causes UB */
  int value = foo();
  return 0;
}
```

When a function is declared to return a value then it has to do so on every possible code path through it. Undefined behavior occurs as soon as the caller (which is expecting a return value) tries to use the return value1.

Note that the undefined behaviour happens *only if* the caller attempts to use/access the value from the function. For example,

```
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* The value (not) returned from foo() is unused. So, this program
   * doesn't cause *undefined behaviour*. */
  foo();
  return 0;
}
```

Version ≥ C99

The `main()` function is an exception to this rule in that it is possible for it to be terminated without a return statement because an assumed return value of `0` will automatically be used in this case2.

1 (*ISO/IEC 9899:201x*, 6.9.1/12)

> If the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

2 (*ISO/IEC 9899:201x*, 5.1.2.2.3/1)

> reaching the } that terminates the main function returns a value of 0.

Section 28.22: Division by zero

```
int x = 0;
```

```c
int y = 5 / x;  /* integer division */
```

or

```c
double x = 0.0;
double y = 5.0 / x;  /* floating point division */
```

or

```c
int x = 0;
int y = 5 % x;  /* modulo operation */
```

For the second line in each example, where the value of the second operand (x) is zero, the behaviour is undefined.

Note that most implementations of floating point math will follow a standard (e.g. IEEE 754), in which case operations like divide-by-zero will have consistent results (e.g., `INFINITY`) even though the C standard says the operation is undefined.

Section 28.23: Conversion between pointer types produces incorrectly aligned result

The following *might* have undefined behavior due to incorrect pointer alignment:

```c
char *memory_block = calloc(sizeof(uint32_t) + 1, 1);
uint32_t *intptr = (uint32_t*)(memory_block + 1);  /* possible undefined behavior */
uint32_t mvalue = *intptr;
```

The undefined behavior happens as the pointer is converted. According to C11, if a *conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3), the behavior is undefined*. Here an `uint32_t` could require alignment of 2 or 4 for example.

`calloc` on the other hand is required to return a pointer that is suitably aligned for any object type; thus `memory_block` is properly aligned to contain an `uint32_t` in its initial part. Then, on a system where `uint32_t` has required alignment of 2 or 4, `memory_block + 1` will be an *odd* address and thus not properly aligned.

Observe that the C standard requests that already the cast operation is undefined. This is imposed because on platforms where addresses are segmented, the byte address `memory_block + 1` may not even have a proper representation as an integer pointer.

Casting `char *` to pointers to other types without any concern to alignment requirements is sometimes incorrectly used for decoding packed structures such as file headers or network packets.

You can avoid the undefined behavior arising from misaligned pointer conversion by using `memcpy`:

```c
memcpy(&mvalue, memory_block + 1, sizeof mvalue);
```

Here no pointer conversion to `uint32_t*` takes place and the bytes are copied one by one.

This copy operation for our example only leads to valid value of `mvalue` because:

- We used `calloc`, so the bytes are properly initialized. In our case all bytes have value `0`, but any other proper initialization would do.
- `uint32_t` is an exact width type and has no padding bits
- Any arbitrary bit pattern is a valid representation for any unsigned type.

Section 28.24: Modifying the string returned by getenv, strerror, and setlocale functions

Modifying the strings returned by the standard functions `getenv()`, `strerror()` and `setlocale()` is undefined. So, implementations may use static storage for these strings.

The getenv() function, C11, §7.22.4.7, 4, says:

> The getenv function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the getenv function.

The strerror() function, C11, §7.23.6.3, 4 says:

> The strerror function returns a pointer to the string, the contents of which are localespecific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the strerror function.

The setlocale() function, C11, §7.11.1.1, 8 says:

> The pointer to string returned by the setlocale function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the setlocale function.

Similarly the `localeconv()` function returns a pointer to `struct lconv` which shall not be modified.

The localeconv() function, C11, §7.11.2.1, 8 says:

> The localeconv function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the localeconv function.

Chapter 29: Random Number Generation

Section 29.1: Basic Random Number Generation

The function `rand()` can be used to generate a pseudo-random integer value between `0` and `RAND_MAX` (`0` and `RAND_MAX` included).

`srand(int)` is used to seed the pseudo-random number generator. Each time `rand()` is seeded wih the same seed, it must produce the same sequence of values. It should only be seeded once before calling `rand()`. It should not be repeatedly seeded, or reseeded every time you wish to generate a new batch of pseudo-random numbers.

Standard practice is to use the result of `time(NULL)` as a seed. If your random number generator requires to have a deterministic sequence, you can seed the generator with the same value on each program start. This is generally not required for release code, but is useful in debug runs to make bugs reproducible.

It is advised to always seed the generator, if not seeded, it behaves as if it was seeded with `srand(1)`.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    int i;
    srand(time(NULL));
    i = rand();

    printf("Random value between [0, %d]: %d\n", RAND_MAX, i);
    return 0;
}
```

Possible output:

```
Random value between [0, 2147483647]: 823321433
```

Notes:

The C Standard does not guarantee the quality of the random sequence produced. In the past, some implementations of `rand()` had serious issues in distribution and randomness of the generated numbers. **The usage of `rand()` is not recommended for serious random number generation needs, like cryptography.**

Section 29.2: Permuted Congruential Generator

Here's a standalone random number generator that doesn't rely on `rand()` or similar library functions.

Why would you want such a thing? Maybe you don't trust your platform's builtin random number generator, or maybe you want a reproducible source of randomness independent of any particular library implementation.

This code is PCG32 from pcg-random.org, a modern, fast, general-purpose RNG with excellent statistical properties. It's not cryptographically secure, so don't use it for cryptography.

```
#include <stdint.h>

/* *Really* minimal PCG32 code / (c) 2014 M.E. O'Neill / pcg-random.org
 * Licensed under Apache License 2.0 (NO WARRANTY, etc. see website) */
```

```
typedef struct { uint64_t state;  uint64_t inc; } pcg32_random_t;

uint32_t pcg32_random_r(pcg32_random_t* rng) {
    uint64_t oldstate = rng->state;
    /* Advance internal state */
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc | 1);
    /* Calculate output function (XSH RR), uses old state for max ILP */
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}

void pcg32_srandom_r(pcg32_random_t* rng, uint64_t initstate, uint64_t initseq) {
    rng->state = 0U;
    rng->inc = (initseq << 1u) | 1u;
    pcg32_random_r(rng);
    rng->state += initstate;
    pcg32_random_r(rng);
}
```

And here's how to call it:

```
#include <stdio.h>
int main(void) {
    pcg32_random_t rng; /* RNG state */
    int i;

    /* Seed the RNG */
    pcg32_srandom_r(&rng, 42u, 54u);

    /* Print some random 32-bit integers */
    for (i = 0; i < 6; i++)
        printf("0x%08x\n", pcg32_random_r(&rng));

    return 0;
}
```

Section 29.3: Xorshift Generation

A good and easy alternative to the flawed `rand()` procedures, is *xorshift*, a class of pseudo-random number generators discovered by George Marsaglia. The xorshift generator is among the fastest non-cryptographically-secure random number generators. More information and other example implementaions are available on the xorshift Wikipedia page

Example implementation

```
#include <stdint.h>

/* These state variables must be initialised so that they are not all zero. */
uint32_t w, x, y, z;

uint32_t xorshift128(void)
{
    uint32_t t = x;
    t ^= t << 11U;
    t ^= t >> 8U;
    x = y; y = z; z = w;
    w ^= w >> 19U;
    w ^= t;
```

```
    return w;
}
```

Section 29.4: Restrict generation to a given range

Usually when generating random numbers it is useful to generate integers within a range, or a p value between 0.0 and 1.0. Whilst modulus operation can be used to reduce the seed to a low integer this uses the low bits, which often go through a short cycle, resulting in a slight skewing of distribution if N is large in proportion to RAND_MAX.

The macro

```
#define uniform() (rand() / (RAND_MAX + 1.0))
```

produces a p value on 0.0 to 1.0 - epsilon, so

```
i = (int)(uniform() * N)
```

will set `i` to a uniform random number within the range 0 to N - 1.

Unfortunately there is a technical flaw, in that RAND_MAX is permitted to be larger than a variable of type `double` can accurately represent. This means that `RAND_MAX + 1.0` evaluates to RAND_MAX and the function occasionally returns unity. This is unlikely however.

Chapter 30: Preprocessor and Macros

All preprocessor commands begins with the hash (pound) symbol #. A C macro is just a preprocessor command that is defined using the `#define` preprocessor directive. During the preprocessing stage, the C preprocessor (a part of the C compiler) simply substitutes the body of the macro wherever its name appears.

Section 30.1: Header Include Guards

Pretty much every header file should follow the include guard idiom:

my-header-file.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// Code body for header file

#endif
```

This ensures that when you `#include "my-header-file.h"` in multiple places, you don't get duplicate declarations of functions, variables, etc. Imagine the following hierarchy of files:

header-1.h

```
typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);
```

header-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

This code has a serious problem: the detailed contents of `MyStruct` is defined twice, which is not allowed. This would result in a compilation error that can be difficult to track down, since one header file includes another. If you instead did it with header guards:

header-1.h

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
```

```
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif
```

header-2.h

```
#ifndef HEADER_2_H
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

This would then expand to:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif

#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // Safe, since HEADER_1_H was #define'd before.
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif

int main() {
    // do something
```

```
}
```

When the compiler reaches the second inclusion of **header-1.h**, `HEADER_1_H` was already defined by the previous inclusion. Ergo, it boils down to the following:

```
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
    // do something
}
```

And thus there is no compilation error.

Note: There are multiple different conventions for naming the header guards. Some people like to name it `HEADER_2_H_`, some include the project name like `MY_PROJECT_HEADER_2_H`. The important thing is to ensure that the convention you follow makes it so that each file in your project has a unique header guard.

If the structure details were not included in the header, the type declared would be incomplete or an opaque type. Such types can be useful, hiding implementation details from users of the functions. For many purposes, the `FILE` type in the standard C library can be regarded as an opaque type (though it usually isn't opaque so that macro implementations of the standard I/O functions can make use of the internals of the structure). In that case, the `header-1.h` could contain:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct MyStruct MyStruct;

int myFunction(MyStruct *value);

#endif
```

Note that the structure must have a tag name (here `MyStruct` — that's in the tags namespace, separate from the ordinary identifiers namespace of the typedef name `MyStruct`), and that the `{ … }` is omitted. This says "there is a structure type `struct MyStruct` and there is an alias for it `MyStruct`".

In the implementation file, the details of the structure can be defined to make the type complete:

```
struct MyStruct {
    …
};
```

If you are using C11, you could repeat the `typedef struct MyStruct MyStruct;` declaration without causing a compilation error, but earlier versions of C would complain. Consequently, it is still best to use the include guard

idiom, even though in this example, it would be optional if the code was only ever compiled with compilers that supported C11.

Many compilers support the `#pragma once` directive, which has the same results:

my-header-file.h

```
#pragma once

// Code for header file
```

However, `#pragma once` is not part of the C standard, so the code is less portable if you use it.

A few headers do not use the include guard idiom. One specific example is the standard **`<assert.h>`** header. It may be included multiple times in a single translation unit, and the effect of doing so depends on whether the macro `NDEBUG` is defined each time the header is included. You may occasionally have an analogous requirement; such cases will be few and far between. Ordinarily, your headers should be protected by the include guard idiom.

Section 30.2: #if 0 to block out code sections

If there are sections of code that you are considering removing or want to temporarily disable, you can comment it out with a block comment.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
*/
```

However, if the source code you have surrounded with a block comment has block style comments in the source, the ending */ of the existing block comments can cause your new block comment to be invalid and cause compilation problems.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;

    /* Return 5 */
    return i;
}
*/
```

In the previous example, the last two lines of the function and the last '*/' are seen by the compiler, so it would compile with errors. A safer method is to use an `#if 0` directive around the code you want to block out.

```
#if 0
/* #if 0 evaluates to false, so everything between here and the #endif are
 * removed by the preprocessor. */
int myUnusedFunction(void)
{
```

```
    int i = 5;
    return i;
}
#endif
```

A benefit with this is that when you want to go back and find the code, it's much easier to do a search for "#if 0" than searching all your comments.

Another very important benefit is that you can nest commenting out code with `#if 0`. This cannot be done with comments.

An alternative to using `#if 0` is to use a name that will not be `#defined` but is more descriptive of why the code is being blocked out. For instance if there is a function that seems to be useless dead code you might use `#if defined(POSSIBLE_DEAD_CODE)` or `#if defined(FUTURE_CODE_REL_020201)` for code needed once other functionality is in place or something similar. Then when going back through to remove or enable that source, those sections of source are easy to find.

Section 30.3: Function-like macros

Function-like macros are similar to **`inline`** functions, these are useful in some cases, such as temporary debug log:

```
#ifdef DEBUG
# define LOGFILENAME "/tmp/logfile.log"

# define LOG(str) do {                              \
  FILE *fp = fopen(LOGFILENAME, "a");              \
  if (fp) {                                         \
    fprintf(fp, "%s:%d %s\n", __FILE__, __LINE__, \
                 /* don't print null pointer */   \
                 str ?str :"<null>");             \
    fclose(fp);                                     \
  }                                                 \
  else {                                            \
    perror("Opening '" LOGFILENAME "' failed");    \
  }                                                 \
} while (0)
#else
  /* Make it a NOOP if DEBUG is not defined. */
# define LOG(LINE) (void)0
#endif


#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
        LOG("There are command line arguments");
    else
        LOG("No command line arguments");
    return 0;
}
```

Here in both cases (with `DEBUG` or not) the call behaves the same way as a function with `void` return type. This ensures that the `if`/`else` conditionals are interpreted as expected.

In the `DEBUG` case this is implemented through a `do { ... } while(0)` construct. In the other case, `(void)0` is a statement with no side effect that is just ignored.

An alternative for the latter would be

```
#define LOG(LINE) do { /* empty */ } while (0)
```

such that it is in all cases syntactically equivalent to the first.

If you use GCC, you can also implement a function-like macro that returns result using a non-standard GNU extension — statement expressions. For example:

```
#include <stdio.h>

#define POW(X, Y) \
({ \
        int i, r = 1; \
        for (i = 0; i < Y; ++i) \
                r *= X; \
        r; \ // returned value is result of last operation
})

int main(void)
{
        int result;

        result = POW(2, 3);
        printf("Result: %d\n", result);
}
```

Section 30.4: Source file inclusion

The most common uses of `#include` preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myheader.h"
```

`#include` replaces the statement with the contents of the file referred to. Angle brackets (<>) refer to header files installed on the system, while quotation marks ("") are for user-supplied files.

Macros themselves can expand other macros once, as this example illustrates:

```
#if VERSION == 1
    #define INCFILE  "vers1.h"
#elif VERSION == 2
    #define INCFILE  "vers2.h"
    /*  and so on */
#else
    #define INCFILE  "versN.h"
#endif
/* ... */
#include INCFILE
```

Section 30.5: Conditional inclusion and conditional function signature modification

To conditionally include a block of code, the preprocessor has several directives (e.g `#if`, `#ifdef`, `#else`, `#endif`, etc).

```
/* Defines a conditional `printf` macro, which only prints if `DEBUG`
```

```
 * has been defined
 */
#ifdef DEBUG
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Normal C relational operators may be used for the `#if` condition

```
#if __STDC_VERSION__ >= 201112L
/* Do stuff for C11 or higher */
#elif __STDC_VERSION__ >= 199901L
/* Do stuff for C99 */
#else
/* Do stuff for pre C99 */
#endif
```

The `#if` directives behaves similar to the C `if` statement, it shall only contain integral constant expressions, and no casts. It supports one additional unary operator, `defined( identifier )`, which returns 1 if the identifier is defined, and `0` otherwise.

```
#if defined(DEBUG) && !defined(QUIET)
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Conditional Function Signature Modification

In most cases a release build of an application is expected to have as little overhead as possible. However during testing of an interim build, additional logs and information about problems found can be helpful.

For example assume there is some function `SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)` which when doing a test build it is desired will generate a log about its use. However this function is used in multiple places and it is desired that when generating the log, part of the information is to know where is the function being called from.

So using conditional compilation you can have something like the following in the include file declaring the function. This replaces the standard version of the function with a debug version of the function. The preprocessor is used to replace calls to the function `SerOpPluAllRead()` with calls to the function `SerOpPluAllRead_Debug()` with two additional arguments, the name of the file and the line number of where the function is used.

Conditional compilation is used to choose whether to override the standard function with a debug version or not.

```
#if 0
// function declaration and prototype for our debug version of the function.
SHORT   SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo);

// macro definition to replace function call using old name with debug function with additional
arguments.
#define SerOpPluAllRead(pPif,usLock) SerOpPluAllRead_Debug(pPif,usLock,__FILE__,__LINE__)
#else
// standard function declaration that is normally used with builds.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd);
#endif
```

This allows you to override the standard version of the function `SerOpPluAllRead()` with a version that will provide the name of the file and line number in the file of where the function is called.

There is one important consideration: *any file using this function must include the header file where this approach is used in order for the preprocessor to modify the function. Otherwise you will see a linker error.*

The definition of the function would look something like the following. What this source does is to request that the preprocessor rename the function `SerOpPluAllRead()` to be `SerOpPluAllRead_Debug()` and to modify the argument list to include two additional arguments, a pointer to the name of the file where the function was called and the line number in the file at which the function is used.

```
#if defined(SerOpPluAllRead)
// forward declare the replacement function which we will call once we create our log.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd);

SHORT    SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo)
{
    int iLen = 0;
    char  xBuffer[256];

    // only print the last 30 characters of the file name to shorten the logs.
    iLen = strlen (aszFilePath);
    if (iLen > 30) {
        iLen = iLen - 30;
    }
    else {
        iLen = 0;
    }

    sprintf (xBuffer, "SerOpPluAllRead_Debug(): husHandle = %d, File %s, lineno = %d",
pPif->husHandle, aszFilePath + iLen, nLineNo);
    IssueDebugLog(xBuffer);

    // now that we have issued the log, continue with standard processing.
    return SerOpPluAllRead_Special(pPif, usLockHnd);
}

// our special replacement function name for when we are generating logs.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd)
#else
// standard, normal function name (signature) that is replaced with our debug version.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)
#endif
{
    if (STUB_SELF == SstReadAsMaster()) {
        return OpPluAllRead(pPif, usLockHnd);
    }
    return OP_NOT_MASTER;
}
```

Section 30.6: __cplusplus for using C externals in C++ code compiled with C++ - name mangling

There are times when an include file has to generate different output from the preprocessor depending on whether the compiler is a C compiler or a C++ compiler due to language differences.

For example a function or other external is defined in a C source file but is used in a C++ source file. Since C++ uses name mangling (or name decoration) in order to generate unique function names based on function argument

types, a C function declaration used in a C++ source file will cause link errors. The C++ compiler will modify the specified external name for the compiler output using the name mangling rules for C++. The result is link errors due to externals not found when the C++ compiler output is linked with the C compiler output.

Since C compilers do not do name mangling but C++ compilers do for all external labels (function names or variable names) generated by the C++ compiler, a predefined preprocessor macro, `__cplusplus`, was introduced to allow for compiler detection.

In order to work around this problem of incompatible compiler output for external names between C and C++, the macro `__cplusplus` is defined in the C++ Preprocessor and is not defined in the C Preprocessor. This macro name can be used with the conditional preprocessor `#ifdef` directive or `#if` with the `defined()` operator to tell whether a source code or include file is being compiled as C++ or C.

```
#ifdef __cplusplus
printf("C++\n");
#else
printf("C\n");
#endif
```

Or you could use

```
#if defined(__cplusplus)
printf("C++\n");
#else
printf("C\n");
#endif
```

In order to specify the correct function name of a function from a C source file compiled with the C compiler that is being used in a C++ source file you could check for the `__cplusplus` defined constant in order to cause the `extern "C" { /* ... */ };` to be used to declare C externals when the header file is included in a C++ source file. However when compiled with a C compiler, the `extern "C" { */ ... */ };` is not used. This conditional compilation is needed because `extern "C" { /* ... */ };` is valid in C++ but not in C.

```
#ifdef __cplusplus
// if we are being compiled with a C++ compiler then declare the
// following functions as C functions to prevent name mangling.
extern "C" {
#endif

// exported C function list.
int foo (void);

#ifdef __cplusplus
// if this is a C++ compiler, we need to close off the extern declaration.
};
#endif
```

Section 30.7: Token pasting

Token pasting allows one to glue together two macro arguments. For example, `front##back` yields `frontback`. A famous example is Win32's **`<TCHAR.H>`** header. In standard C, one can write `L"string"` to declare a wide character string. However, Windows API allows one to convert between wide character strings and narrow character strings simply by `#define`ing `UNICODE`. In order to implement the string literals, `TCHAR.H` uses this

```
#ifdef UNICODE
#define TEXT(x) L##x
```

```
#endif
```

Whenever a user writes `TEXT("hello, world")`, and UNICODE is defined, the C preprocessor concatenates `L` and the macro argument. `L` concatenated with `"hello, world"` gives `L"hello, world"`.

Section 30.8: Predefined Macros

A predefined macro is a macro that is already understood by the C pre processor without the program needing to define it. Examples include

Mandatory Pre-Defined Macros

- `__FILE__`, which gives the file name of the current source file (a string literal),
- `__LINE__` for the current line number (an integer constant),
- `__DATE__` for the compilation date (a string literal),
- `__TIME__` for the compilation time (a string literal).

There's also a related predefined identifier, `__func__` (ISO/IEC 9899:2011 §6.4.2.2), which is *not* a macro:

> The identifier `__func__` shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration:
>
> ```
> static const char __func__[] = "function-name";
> ```
>
> appeared, where *function-name* is the name of the lexically-enclosing function.

`__FILE__`, `__LINE__` and `__func__` are especially useful for debugging purposes. For example:

```
fprintf(stderr, "%s: %s: %d: Denominator is 0", __FILE__, __func__, __LINE__);
```

Pre-C99 compilers, may or may not support `__func__` or may have a macro that acts the same that is named differently. For example, gcc used `__FUNCTION__` in C89 mode.

The below macros allow to ask for detail on the implementation:

- `__STDC_VERSION__` The version of the C Standard implemented. This is a constant integer using the format `yyyymmL` (the value `201112L` for C11, the value `199901L` for C99; it wasn't defined for C89/C90)
- `__STDC_HOSTED__` `1` if it's a hosted implementation, else `0`.
- `__STDC__` If `1`, the implementation conforms to the C Standard.

Other Pre-Defined Macros (non mandatory)

ISO/IEC 9899:2011 §6.10.9.2 Environment macros:

> - `__STDC_ISO_10646__` An integer constant of the form `yyyymmL` (for example, 199712L). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type `wchar_t`, has the same value as the short identifier of that character. The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.
>
> - `__STDC_MB_MIGHT_NEQ_WC__` The integer constant 1, intended to indicate that, in the encoding for `wchar_t`, a member of the basic character set need not have a code value equal to its value when

> used as the lone character in an integer character constant.
>
> - `__STDC_UTF_16__` The integer constant 1, intended to indicate that values of type `char16_t` are UTF−16 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.
> - `__STDC_UTF_32__` The integer constant 1, intended to indicate that values of type `char32_t` are UTF−32 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

ISO/IEC 9899:2011 §6.10.8.3 Conditional feature macros

> - `__STDC_ANALYZABLE__` The integer constant 1, intended to indicate conformance to the specifications in annex L (Analyzability).
> - `__STDC_IEC_559__` The integer constant 1, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic).
> - `__STDC_IEC_559_COMPLEX__` The integer constant 1, intended to indicate adherence to the specifications in annex G (IEC 60559 compatible complex arithmetic).
> - `__STDC_LIB_EXT1__` The integer constant `201112L`, intended to indicate support for the extensions defined in annex K (Bounds-checking interfaces).
> - `__STDC_NO_ATOMICS__` The integer constant 1, intended to indicate that the implementation does not support atomic types (including the `_Atomic` type qualifier) and the **`<stdatomic.h>`** header.
> - `__STDC_NO_COMPLEX__` The integer constant 1, intended to indicate that the implementation does not support complex types or the **`<complex.h>`** header.
> - `__STDC_NO_THREADS__` The integer constant 1, intended to indicate that the implementation does not support the **`<threads.h>`** header.
> - `__STDC_NO_VLA__` The integer constant 1, intended to indicate that the implementation does not support variable length arrays or variably modified types.

Section 30.9: Variadic arguments macro

Version ≥ C99

Macros with variadic args:

Let's say you want to create some print-macro for debugging your code, let's take this macro as an example:

```
#define debug_print(msg) printf("%s:%d %s", __FILE__, __LINE__, msg)
```

Some examples of usage:

The function `somefunc()` returns -1 if failed and 0 if succeeded, and it is called from plenty different places within the code:

```
int retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}

/* some other code */
```

```
 retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}
```

What happens if the implementation of `somefunc()` changes, and it now returns different values matching different possible error types? You still want use the debug macro and print the error value.

```
debug_printf(retVal);      /* this would obviously fail */
debug_printf("%d",retVal); /* this would also fail */
```

To solve this problem the `__VA_ARGS__` macro was introduced. This macro allows multiple parameters X-macro's:

Example:

```
 #define debug_print(msg, ...) printf(msg, __VA_ARGS__) \
                               printf("\nError occurred in file:line (%s:%d)\n", __FILE__, __LINE)
```

Usage:

```
int retVal = somefunc();

debug_print("retVal of somefunc() is-> %d", retVal);
```

This macro allows you to pass multiple parameters and print them, but now it forbids you from sending any parameters at all.

```
debug_print("Hey");
```

This would raise some syntax error as the macro expects at least one more argument and the pre-processor would not ignore the lack of comma in the `debug_print()` macro. Also `debug_print("Hey",);` would raise a syntax error as you cant keep the argument passed to macro empty.

To solve this, `##__VA_ARGS__` macro was introduced, this macro states that if no variable arguments exist, the comma is deleted by the pre-processor from code.

Example:

```
 #define debug_print(msg, ...) printf(msg, ##__VA_ARGS__) \
                               printf("\nError occurred in file:line (%s:%d)\n", __FILE__, __LINE)
```

Usage:

```
 debug_print("Ret val of somefunc()?");
 debug_print("%d",somefunc());
```

Section 30.10: Macro Replacement

The simplest form of macro replacement is to define a `manifest constant`, as in

```
#define ARRSIZE 100
int array[ARRSIZE];
```

This defines a *function-like* macro that multiplies a variable by `10` and stores the new value:

```
#define TIMES10(A) ((A) *= 10)

double b = 34;
int c = 23;

TIMES10(b);   // good: ((b) *= 10);
TIMES10(c);   // good: ((c) *= 10);
TIMES10(5);   // bad:  ((5) *= 10);
```

The replacement is done before any other interpretation of the program text. In the first call to `TIMES10` the name `A` from the definition is replaced by `b` and the so expanded text is then put in place of the call. Note that this definition of `TIMES10` is not equivalent to

```
#define TIMES10(A) ((A) = (A) * 10)
```

because this could evaluate the replacement of `A`, twice, which can have unwanted side effects.

The following defines a function-like macro which value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and of generating more code than a function if invoked several times.

```
#define max(a, b) ((a) > (b) ? (a) : (b))

int maxVal = max(11, 43);              /* 43 */
int maxValExpr = max(11 + 36, 51 - 7); /* 47 */

/* Should not be done, due to expression being evaluated twice */
int j = 0, i = 0;
int sideEffect = max(++i, ++j);       /* i == 4 */
```

Because of this, such macros that evaluate their arguments multiple times are usually avoided in production code. Since C11 there is the `_Generic` feature that allows to avoid such multiple invocations.

The abundant parentheses in the macro expansions (right hand side of the definition) ensure that the arguments and the resulting expression are bound properly and fit well into the context in which the macro is called.

Section 30.11: Error directive

If the preprocessor encounters an `#error` directive, compilation is halted and the diagnostic message included is printed.

```
#define DEBUG

#ifdef DEBUG
#error "Debug Builds Not Supported"
#endif

int main(void) {
    return 0;
}
```

Possible output:

```
$ gcc error.c
error.c: error: #error "Debug Builds Not Supported"
```

Section 30.12: FOREACH implementation

We can also use macros for making code easier to read and write. For example we can implement macros for implementing the `foreach` construct in C for some data structures like singly- and doubly-linked lists, queues, etc.

Here is a small example.

```
#include <stdio.h>
#include <stdlib.h>

struct LinkedListNode
{
    int data;
    struct LinkedListNode *next;
};

#define FOREACH_LIST(node, list) \
     for (node=list; node; node=node->next)

/* Usage */
int main(void)
{
    struct LinkedListNode *list, **plist = &list, *node;
    int i;

    for (i=0; i<10; i++)
    {
         *plist = malloc(sizeof(struct LinkedListNode));
         (*plist)->data = i;
         (*plist)->next = NULL;
         plist          = &(*plist)->next;
    }

    /* printing the elements here */
    FOREACH_LIST(node, list)
    {
        printf("%d\n", node->data);
    }
}
```

You can make a standard interface for such data-structures and write a generic implementation of `FOREACH` as:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct CollectionItem_
{
    int data;
    struct CollectionItem_ *next;
} CollectionItem;

typedef struct Collection_
{
    /* interface functions */
    void* (*first)(void *coll);
    void* (*last) (void *coll);
    void* (*next) (void *coll, CollectionItem *currItem);
```

```c
    CollectionItem *collectionHead;
    /* Other fields */
} Collection;

/* must implement */
void *first(void *coll)
{
    return ((Collection*)coll)->collectionHead;
}

/* must implement */
void *last(void *coll)
{
    return NULL;
}

/* must implement */
void *next(void *coll, CollectionItem *curr)
{
    return curr->next;
}

CollectionItem *new_CollectionItem(int data)
{
    CollectionItem *item = malloc(sizeof(CollectionItem));
    item->data = data;
    item->next = NULL;
    return item;
}

void Add_Collection(Collection *coll, int data)
{
    CollectionItem **item = &coll->collectionHead;
    while(*item)
        item = &(*item)->next;
    (*item) = new_CollectionItem(data);
}

Collection *new_Collection()
{
    Collection *nc = malloc(sizeof(Collection));
    nc->first = first;
    nc->last  = last;
    nc->next  = next;
    return nc;
}

/* generic implementation */
#define FOREACH(node, collection)                      \
    for (node  = (collection)->first(collection);      \
         node != (collection)->last(collection);       \
         node  = (collection)->next(collection, node))

int main(void)
{
    Collection *coll = new_Collection();
    CollectionItem *node;
    int i;

    for(i=0; i<10; i++)
    {
         Add_Collection(coll, i);
```

```
    }

    /* printing the elements here */
    FOREACH(node, coll)
    {
        printf("%d\n", node->data);
    }
}
```

To use this generic implementation just implement these functions for your data structure.

```
void* (*first)(void *coll);
void* (*last) (void *coll);
void* (*next) (void *coll, CollectionItem *currItem);
```

Chapter 31: Signal handling

Parameter	Details
sig	The signal to set the signal handler to, one of `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGTERM`, `SIGINT`, `SIGSEGV` or some implementation defined value
func	The signal handler, which is either of the following: `SIG_DFL`, for the default handler, `SIG_IGN` to ignore the signal, or a function pointer with the signature `void foo(int sig);`.

Section 31.1: Signal Handling with “signal()”

Signal numbers can be synchronous (like `SIGSEGV` – segmentation fault) when they are triggered by a malfunctioning of the program itself or asynchronous (like `SIGINT` - interactive attention) when they are initiated from outside the program, e.g by a keypress as `Cntrl-C`.

The `signal()` function is part of the ISO C standard and can be used to assign a function to handle a specific signal

```
#include <stdio.h>  /* printf() */
#include <stdlib.h> /* abort()  */
#include <signal.h> /* signal() */

void handler_nonportable(int sig)
{
    /* undefined behavior, maybe fine on specific platform */
    printf("Catched: %d\n", sig);

    /* abort is safe to call */
    abort();
}

sig_atomic_t volatile finished = 0;

void handler(int sig)
{
    switch (sig) {
    /* hardware interrupts should not return */
    case SIGSEGV:
    case SIGFPE:
    case SIGILL:
```

Version ≥ C11

```
      /* quick_exit is safe to call */
      quick_exit(EXIT_FAILURE);
```

Version < C11

```
      /* use _Exit in pre-C11 */
      _Exit(EXIT_FAILURE);
```

```
    default:
       /* Reset the signal to the default handler,
          so we will not be called again if things go
          wrong on return. */
      signal(sig, SIG_DFL);
      /* let everybody know that we are finished */
      finished = sig;
      return;
   }
}

int main(void)
{
```

```c
    /* Catch the SIGSEGV signal, raised on segmentation faults (i.e NULL ptr access */
    if (signal(SIGSEGV, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGSEGV");
        return EXIT_FAILURE;
    }

    /* Catch the SIGTERM signal, termination request */
    if (signal(SIGTERM, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGTERM");
        return EXIT_FAILURE;
    }

    /* Ignore the SIGINT signal, by setting the handler to `SIG_IGN`. */
    signal(SIGINT, SIG_IGN);


    /* Do something that takes some time here, and leaves
       the time to terminate the program from the keyboard. */

    /* Then: */

    if (finished) {
       fprintf(stderr, "we have been terminated by signal %d\n", (int)finished);
        return EXIT_FAILURE;
    }


    /* Try to force a segmentation fault, and raise a SIGSEGV */
    {
      char* ptr = 0;
      *ptr = 0;
    }

    /* This should never be executed */
    return EXIT_SUCCESS;
}
```

Using `signal()` imposes important limitations what you are allowed to do inside the signal handlers, see the remarks for further information.

POSIX recommends the usage of `sigaction()` instead of `signal()`, due to its underspecified behavior and significant implementation variations. POSIX also defines many more signals than ISO C standard, including `SIGUSR1` and `SIGUSR2`, which can be used freely by the programmer for any purpose.

Chapter 32: Variable arguments

Parameter	Details
`va_list` ap	argument pointer, current position in the list of variadic arguments
last	name of last non-variadic function argument, so the compiler finds the correct place to start processing variadic arguments; may not be declared as a `register` variable, a function, or an array type
type	**promoted** type of the variadic argument to read (e.g. `int` for a `short int` argument)
`va_list` src	current argument pointer to copy
`va_list` dst	new argument list to be filled in

Variable arguments are used by functions in the printf family (`printf`, `fprintf`, etc) and others to allow a function to be called with a different number of arguments each time, hence the name *varargs*.

To implement functions using the variable arguments feature, use `#include <stdarg.h>`.

To call functions which take a variable number of arguments, ensure there is a full prototype with the trailing ellipsis in scope: `void err_exit(const char *format, ...);` for example.

Section 32.1: Using an explicit count argument to determine the length of the va_list

With any variadic function, the function must know how to interpret the variable arguments list. With the `printf()` or `scanf()` functions, the format string tells the function what to expect.

The simplest technique is to pass an explicit count of the other arguments (which are normally all the same type). This is demonstrated in the variadic function in the code below which calculates the sum of a series of integers, where there may be any number of integers but that count is specified as an argument prior to the variable argument list.

```
#include <stdio.h>
#include <stdarg.h>

/* first arg is the number of following int args to sum. */
int sum(int n, ...) {
    int sum = 0;
    va_list it; /* hold information about the variadic argument list. */

    va_start(it, n); /* start variadic argument processing */
    while (n--)
      sum += va_arg(it, int); /* get and sum the next variadic argument */
    va_end(it); /* end variadic argument processing */

    return sum;
}

int main(void)
{
    printf("%d\n", sum(5, 1, 2, 3, 4, 5)); /* prints 15 */
    printf("%d\n", sum(10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /* prints 666 */
    return 0;
}
```

Section 32.2: Using terminator values to determine the end of va_list

With any variadic function, the function must know how to interpret the variable arguments list. The "traditional" approach (exemplified by `printf`) is to specify number of arguments up front. However, this is not always a good idea:

```
/* First argument specifies the number of parameters; the remainder are also int */
extern int sum(int n, ...);

/* But it's far from obvious from the code. */
sum(5, 2, 1, 4, 3, 6)

/* What happens if i.e. one argument is removed later on? */
sum(5, 2, 1, 3, 6)  /* Disaster */
```

Sometimes it's more robust to add an explicit terminator, exemplified by the POSIX `execlp()` function. Here's another function to calculate the sum of a series of `double` numbers:

```
#include <stdarg.h>
#include <stdio.h>
#include <math.h>

/* Sums args up until the terminator NAN */
double sum (double x, ...) {
  double sum = 0;
  va_list va;

  va_start(va, x);
  for (; !isnan(x); x = va_arg(va, double)) {
    sum += x;
  }
  va_end(va);

  return sum;
}

int main (void) {
  printf("%g\n", sum(5., 2., 1., 4., 3., 6., NAN));
  printf("%g\n", sum(1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));
}
```

Good terminator values:

- integer (supposed to be all positive or non-negative) — `0` or `-1`
- floating point types — `NAN`
- pointer types — `NULL`
- enumerator types — some special value

Section 32.3: Implementing functions with a `printf()`-like interface

One common use of variable-length argument lists is to implement functions that are a thin wrapper around the `printf()` family of functions. One such example is a set of error reporting functions.

errmsg.h

```c
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

void verrmsg(int errnum, const char *fmt, va_list ap);
noreturn void errmsg(int exitcode, int errnum, const char *fmt, ...);
void warnmsg(int errnum, const char *fmt, ...);

#endif
```

This is a bare-bones example; such packages can be much elaborate. Normally, programmers will use either `errmsg()` or `warnmsg()`, which themselves use `verrmsg()` internally. If someone comes up with a need to do more, though, then the exposed `verrmsg()` function will be useful. You could avoid exposing it until you have a need for it (YAGNI — you aren't gonna need it), but the need will arise eventually (you *are* gonna need it — YAGNI).

`errmsg.c`

This code only needs to forward the variadic arguments to the `vfprintf()` function for outputting to standard error. It also reports the system error message corresponding to the system error number (`errno`) passed to the functions.

```c
#include "errmsg.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
verrmsg(int errnum, const char *fmt, va_list ap)
{
    if (fmt)
        vfprintf(stderr, fmt, ap);
    if (errnum != 0)
        fprintf(stderr, ": %s", strerror(errnum));
    putc('\n', stderr);
}

void
errmsg(int exitcode, int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
    exit(exitcode);
}

void
warnmsg(int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
}
```

Using `errmsg.h`

Now you can use those functions as follows:

```
#include "errmsg.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char buffer[BUFSIZ];
    int fd;
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *filename = argv[1];

    if ((fd = open(filename, O_RDONLY)) == -1)
        errmsg(EXIT_FAILURE, errno, "cannot open %s", filename);
    if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))
        errmsg(EXIT_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer), filename);
    if (close(fd) == -1)
        warnmsg(errno, "cannot close %s", filename);
    /* continue the program */
    return 0;
}
```

If either the `open()` or `read()` system calls fails, the error is written to standard error and the program exits with exit code 1. If the `close()` system call fails, the error is merely printed as a warning message, and the program continues.

Checking the correct use of `printf()` formats

If you are using GCC (the GNU C Compiler, which is part of the GNU Compiler Collection), or using Clang, then you can have the compiler check that the arguments you pass to the error message functions match what `printf()` expects. Since not all compilers support the extension, it needs to be compiled conditionally, which is a little bit fiddly. However, the protection it gives is worth the effort.

First, we need to know how to detect that the compiler is GCC or Clang emulating GCC. The answer is that GCC defines `__GNUC__` to indicate that.

See common function attributes for information about the attributes — specifically the `format` attribute.

Rewritten `errmsg.h`

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

#if !defined(PRINTFLIKE)
#if defined(__GNUC__)
#define PRINTFLIKE(n,m) __attribute__((format(printf,n,m)))
#else
#define PRINTFLIKE(n,m) /* If only */
```

```
#endif /* __GNUC__ */
#endif /* PRINTFLIKE */

void verrmsg(int errnum, const char *fmt, va_list ap);
void noreturn errmsg(int exitcode, int errnum, const char *fmt, ...)
        PRINTFLIKE(3, 4);
void warnmsg(int errnum, const char *fmt, ...)
        PRINTFLIKE(2, 3);

#endif
```

Now, if you make a mistake like:

```
errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
```

(where the `%d` should be `%s`), then the compiler will complain:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \
>     -Wold-style-definition -c erruse.c
erruse.c: In function 'main':
erruse.c:20:64: error: format '%d' expects argument of type 'int', but argument 4 has type 'const char
*' [-Werror=format=]
         errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
                                                           ~^
                                                           %s
cc1: all warnings being treated as errors
$
```

Section 32.4: Using a format string

Using a format string provides information about the expected number and type of the subsequent variadic arguments in such a way as to avoid the need for an explicit count argument or a terminator value.

The example below shows a a function that wraps the standard `printf()` function, only allowing for the use of variadic arguments of the type `char`, `int` and `double` (in decimal floating point format). Here, like with `printf()`, the first argument to the wrapping function is the format string. As the format string is parsed the function is able to determine if there is another variadic argument expected and what it's type should be.

```
#include <stdio.h>
#include <stdarg.h>

int simple_printf(const char *format, ...)
{
    va_list ap; /* hold information about the variadic argument list. */
    int printed = 0; /* count of printed characters */

    va_start(ap, format); /* start variadic argument processing */

    while (*format != '\0') /* read format string until string terminator */
    {
        int f = 0;

        if (*format == '%')
        {
            ++format;
            switch(*format)
            {
                case 'c' :
```

```
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument, note type
promotion from char to int */
                    break;
                case 'd' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument */
                    break;

                case 'f' :
                    f = printf("%f", va_arg(ap, double)); /* print next variadic argument */
                    break;
                default :
                    f = -1; /* invalid format specifier */
                    break;
            }
        }
        else
        {
            f = printf("%c", *format); /* print any other characters */
        }

        if (f < 0) /* check for errors */
        {
            printed = f;
            break;
        }
        else
        {
            printed += f;
        }
        ++format; /* move on to next character in string */
    }

    va_end(ap); /* end variadic argument processing */

    return printed;
}

int main (int argc, char *argv[])
{
    int x = 40;
    int y = 0;

    y = simple_printf("There are %d characters in this sentence", x);
    simple_printf("\n%d were printed\n", y);
}
```

Chapter 33: Assertion

Parameter	Details
expression	expression of scalar type.
message	string literal to be included in the diagnostic message.

An ***assertion*** is a predicate that the presented condition must be true at the moment the assertion is encountered by the software. Most common are ***simple assertions***, which are validated at execution time. However, ***static assertions*** are checked at compile time.

Section 33.1: Simple Assertion

An assertion is a statement used to assert that a fact must be true when that line of code is reached. Assertions are useful for ensuring that expected conditions are met. When the condition passed to an assertion is true, there is no action. The behavior on false conditions depends on compiler flags. When assertions are enabled, a false input causes an immediate program halt. When they are disabled, no action is taken. It is common practice to enable assertions in internal and debug builds, and disable them in release builds, though assertions are often enabled in release. (Whether termination is better or worse than errors depends on the program.) Assertions should be used only to catch internal programming errors, which usually means being passed bad parameters.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int main(void)
{
    int x = -1;
    assert(x >= 0);

    printf("x = %d\n", x);
    return 0;
}
```

Possible output with `NDEBUG` undefined:

```
a.out: main.c:9: main: Assertion `x >= 0' failed.
```

Possible output with `NDEBUG` defined:

```
x = -1
```

It's good practice to define `NDEBUG` globally, so that you can easily compile your code with all assertions either on or off. An easy way to do this is define `NDEBUG` as an option to the compiler, or define it in a shared configuration header (e.g. `config.h`).

Section 33.2: Static Assertion

Version ≥ C11

Static assertions are used to check if a condition is true when the code is compiled. If it isn't, the compiler is required to issue an error message and stop the compiling process.

A static assertion is one that is checked at compile time, not run time. The condition must be a constant expression, and if false will result in a compiler error. The first argument, the condition that is checked, must be a constant expression, and the second a string literal.

Unlike assert, `_Static_assert` is a keyword. A convenience macro `static_assert` is defined in **`<assert.h>`**.

```
#include <assert.h>

enum {N = 5};
_Static_assert(N == 5, "N does not equal 5");
static_assert(N > 10, "N is not greater than 10");  /* compiler error */
```

Version = C99

Prior to C11, there was no direct support for static assertions. However, in C99, static assertions could be emulated with macros that would trigger a compilation failure if the compile time condition was false. Unlike `_Static_assert`, the second parameter needs to be a proper token name so that a variable name can be created with it. If the assertion fails, the variable name is seen in the compiler error, since that variable was used in a syntactically incorrect array declaration.

```
#define STATIC_MSG(msg, l) STATIC_MSG2(msg, l)
#define STATIC_MSG2(msg,l) on_line_##l##__##msg
#define STATIC_ASSERT(x, msg) extern char STATIC_MSG(msg, __LINE__) [(x)?1:-1]

enum { N = 5 };
STATIC_ASSERT(N == 5, N_must_equal_5);
STATIC_ASSERT(N > 5, N_must_be_greater_than_5); /* compile error */
```

Before C99, you could not declare variables at arbitrary locations in a block, so you would have to be extremely cautious about using this macro, ensuring that it only appears where a variable declaration would be valid.

Section 33.3: Assert Error Messages

A trick exists that can display an error message along with an assertion. Normally, you would write code like this

```
void f(void *p)
{
    assert(p != NULL);
    /* more code */
}
```

If the assertion failed, an error message would resemble

> Assertion failed: p != NULL, file main.c, line 5

However, you can use logical AND (`&&`) to give an error message as well

```
void f(void *p)
{
    assert(p != NULL && "function f: p cannot be NULL");
    /* more code */
}
```

Now, if the assertion fails, an error message will read something like this

> Assertion failed: p != NULL && "function f: p cannot be NULL", file main.c, line 5

The reason as to why this works is that a string literal always evaluates to non-zero (true). Adding `&& 1` to a Boolean expression has no effect. Thus, adding `&& "error message"` has no effect either, except that the compiler will display the entire expression that failed.

Section 33.4: Assertion of Unreachable Code

During development, when certain code paths must be prevented from the reach of control flow, you may use `assert(0)` to indicate that such a condition is erroneous:

```
switch (color) {
    case COLOR_RED:
    case COLOR_GREEN:
    case COLOR_BLUE:
        break;

    default:
        assert(0);
}
```

Whenever the argument of the `assert()` macro evaluates false, the macro will write diagnostic information to the standard error stream and then abort the program. This information includes the file and line number of the `assert()` statement and can be very helpful in debugging. Asserts can be disabled by defining the macro `NDEBUG`.

Another way to terminate a program when an error occurs are with the standard library functions `exit`, `quick_exit` or `abort`. `exit` and `quick_exit` take an argument that can be passed back to your environment. `abort()` (and thus `assert`) can be a really severe termination of your program, and certain cleanups that would otherwise be performed at the end of the execution, may not be performed.

The primary advantage of `assert()` is that it automatically prints debugging information. Calling `abort()` has the advantage that it cannot be disabled like an assert, but it may not cause any debugging information to be displayed. In some situations, using both constructs together may be beneficial:

```
if (color == COLOR_RED || color == COLOR_GREEN) {
   ...
} else if (color == COLOR_BLUE) {
   ...
} else {
   assert(0), abort();
}
```

When asserts are *enabled*, the `assert()` call will print debug information and terminate the program. Execution never reaches the `abort()` call. When asserts are *disabled*, the `assert()` call does nothing and `abort()` is called. This ensures that the program *always* terminates for this error condition; enabling and disabling asserts only effects whether or not debug output is printed.

You should never leave such an `assert` in production code, because the debug information is not helpful for end users and because `abort` is generally a much too severe termination that inhibit cleanup handlers that are installed for `exit` or `quick_exit` to run.

Section 33.5: Precondition and Postcondition

One use case for assertion is precondition and postcondition. This can be very useful to maintain invariant and

design by contract. For a example a length is always zero or positive so this function must return a zero or positive value.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int length2 (int *a, int count)
{
    int i, result = 0;

    /* Precondition: */
    /* NULL is an invalid vector */
    assert (a != NULL);
    /* Number of dimensions can not be negative.*/
    assert (count >= 0);

    /* Calculation */
    for (i = 0; i < count; ++i)
    {
        result = result + (a[i] * a[i]);
    }

    /* Postcondition: */
    /* Resulting length can not be negative. */
    assert (result >= 0);
    return result;
}

#define COUNT 3

int main (void)
{
    int a[COUNT] = {1, 2, 3};
    int *b = NULL;
    int r;
    r = length2 (a, COUNT);
    printf ("r = %i\n", r);
    r = length2 (b, COUNT);
    printf ("r = %i\n", r);
    return 0;
}
```

Chapter 34: Generic selection

Parameter	Details
generic-assoc-list	generic-association **OR** generic-assoc-list , generic-association
generic-association	type-name : assignment-expression **OR** default : assignment-expression

Section 34.1: Check whether a variable is of a certain qualified type

```
#include <stdio.h>

#define is_const_int(x) _Generic((&x),  \
        const int *: "a const int",     \
        int *:       "a non-const int", \
        default:     "of other type")

int main(void)
{
    const int i = 1;
    int j = 1;
    double k = 1.0;
    printf("i is %s\n", is_const_int(i));
    printf("j is %s\n", is_const_int(j));
    printf("k is %s\n", is_const_int(k));
}
```

Output:

```
i is a const int
j is a non-const int
k is of other type
```

However, if the type generic macro is implemented like this:

```
#define is_const_int(x) _Generic((x), \
        const int: "a const int",     \
        int:       "a non-const int", \
        default:   "of other type")
```

The output is:

```
i is a non-const int
j is a non-const int
k is of other type
```

This is because all type qualifiers are dropped for the evaluation of the controlling expression of a `_Generic` primary expression.

Section 34.2: Generic selection based on multiple arguments

If a selection on multiple arguments for a type generic expression is wanted, and all types in question are arithmetic types, an easy way to avoid nested `_Generic` expressions is to use addition of the parameters in the controlling expression:

```c
int max_int(int, int);
unsigned max_unsigned(unsigned, unsigned);
double max_double(double, double);

#define MAX(X, Y) _Generic((X)+(Y),                \
                           int:      max_int,      \
                           unsigned: max_unsigned, \
                           default:  max_double)   \
                     ((X), (Y))
```

Here, the controlling expression `(X)+(Y)` is only inspected according to its type and not evaluated. The usual conversions for arithmetic operands are performed to determine the selected type.

For more complex situation, a selection can be made based on more than one argument to the operator, by nesting them together.

This example selects between four externally implemented functions, that take combinations of two int and/or string arguments, and return their sum.

```c
int AddIntInt(int a, int b);
int AddIntStr(int a, const char* b);
int AddStrInt(const char*  a, int b );
int AddStrStr(const char*  a, const char*  b);

#define AddStr(y)                            \
   _Generic((y),         int: AddStrInt,     \
                        char*: AddStrStr,    \
                  const char*: AddStrStr )

#define AddInt(y)                            \
   _Generic((y),          int: AddIntInt,    \
                        char*: AddIntStr,    \
                  const char*: AddIntStr )

#define Add(x, y)                            \
   _Generic((x) ,        int: AddInt(y) ,    \
                       char*: AddStr(y) ,    \
                 const char*: AddStr(y))     \
                        ((x), (y))

int main( void )
{
    int result = 0;
    result = Add( 100 , 999 );
    result = Add( 100 , "999" );
    result = Add( "100" , 999 );
    result = Add( "100" , "999" );

    const int a = -123;
    char b[] = "4321";
    result = Add( a , b );

    int c = 1;
    const char d[] = "0";
    result = Add( d , ++c );
}
```

Even though it appears as if argument y is evaluated more than once, it isn't 1. Both arguments are evaluated only once, at the end of macro Add: `( x , y )`, just like in an ordinary function call.

1 (Quoted from: ISO:IEC 9899:201X 6.5.1.1 Generic selection 3)
The controlling expression of a generic selection is not evaluated.

Section 34.3: Type-generic printing macro

```
#include <stdio.h>

void print_int(int x) { printf("int: %d\n", x); }
void print_dbl(double x) { printf("double: %g\n", x); }
void print_default() { puts("unknown argument"); }

#define print(X) _Generic((X), \
        int: print_int, \
        double: print_dbl, \
        default: print_default)(X)

int main(void) {
    print(42);
    print(3.14);
    print("hello, world");
}
```

Output:

```
int: 42
double: 3.14
unknown argument
```

Note that if the type is neither `int` nor `double`, a warning would be generated. To eliminate the warning, you can add that type to the `print(X)` macro.

Chapter 35: X-macros

X-macros are a preprocessor-based technique for minimizing repetitious code and maintaining data / code correspondences. Multiple distinct macro expansions based on a common set of data are supported by representing the whole group of expansions via a single master macro, with that macro's replacement text consisting of a sequence of expansions of an inner macro, one for each datum. The inner macro is traditionally named `X()`, hence the name of the technique.

Section 35.1: Trivial use of X-macros for printfs

```
/* define a list of preprocessor tokens on which to call X */
#define X_123 X(1) X(2) X(3)

/* define X to use */
#define X(val) printf("X(%d) made this print\n", val);
X_123
#undef X
/* good practice to undef X to facilitate reuse later on */
```

This example will result in the preprocessor generating the following code:

```
printf("X(%d) made this print\n", 1);
printf("X(%d) made this print\n", 2);
printf("X(%d) made this print\n", 3);
```

Section 35.2: Extension: Give the X macro as an argument

The X-macro approach can be generalized a bit by making the name of the "X" macro an argument of the master macro. This has the advantages of helping to avoid macro name collisions and of allowing use of a general-purpose macro as the "X" macro.

As always with X macros, the master macro represents a list of items whose significance is specific to that macro. In this variation, such a macro might be defined like so:

```
/* declare list of items */
#define ITEM_LIST(X) \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */
```

One might then generate code to print the item names like so:

```
/* define macro to apply */
#define PRINTSTRING(value) printf( #value "\n");

/* apply macro to the list of items */
ITEM_LIST(PRINTSTRING)
```

That expands to this code:

```
printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n");
```

In contrast to standard X macros, where the "X" name is a built-in characteristic of the master macro, with this style

it may be unnecessary or even undesirable to afterward undefine the macro used as the argument (PRINTSTRING in this example).

Section 35.3: Enum Value and Identifier

```
/* declare items of the enum */
#define FOREACH \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */

/* define the enum values */
#define X(id) MyEnum_ ## id,
enum MyEnum { FOREACH };
#undef X

/* convert an enum value to its identifier */
const char * enum2string(int enumValue)
{
    const char* stringValue = NULL;
#define X(id) if (enumValue == MyEnum_ ## id) stringValue = #id;
    FOREACH
#undef X
    return stringValue;
}
```

Next you can use the enumerated value in your code and easily print its identifier using :

```
printf("%s\n", enum2string(MyEnum_item2));
```

Section 35.4: Code generation

X-Macros can be used for code generation, by writing repetitive code: iterate over a list to do some tasks, or to declare a set of constants, objects or functions.

Here we use X-macros to declare an enum containing 4 commands and a map of their names as strings

Then we can print the string values of the enum.

```
/* All our commands */
#define COMMANDS(OP) OP(Open) OP(Close) OP(Save) OP(Quit)

/* generate the enum Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; */
#define ENUM_NAME(name) cmd##name,
enum Commands {
  COMMANDS(ENUM_NAME)
};
#undef ENUM_NAME

/* generate the string table */
#define COMMAND_OP(name) #name,
const char* const commandNames[] = {
  COMMANDS(COMMAND_OP)
};
#undef COMMAND_OP

/* the following prints "Quit\n": */
```

```
printf("%s\n", commandNames[cmdQuit]());
```

Similarly, we can generate a jump table to call functions by the enum value.

This requires all functions to have the same signature. If they take no arguments and return an int, we would put this in a header with the enum definition:

```
/* declare all functions as extern */
#define EXTERN_FUNC(name) extern int doCmd##name(void);
COMMANDS(EXTERN_FUNC)
#undef EXTERN_FUNC

/* declare the function pointer type and the jump table  */
typedef int (*CommandFunc)(void);
extern CommandFunc commandJumpTable[];
```

All of the following can be in different compilation units assuming the part above is included as a header:

```
/* generate the jump table */
#define FUNC_NAME(name) doCmd##name,
CommandFunc commandJumpTable[] = {
  COMMANDS(FUNC_NAME)
};
#undef FUNC_NAME

/* call the save command like this: */
int result = commandJumpTable[cmdSave]();

/* somewhere else, we need the implementations of the commands */
int doCmdOpen(void) {/* code performing open command */}
int doCmdClose(void) {/* code performing close command */}
int doCmdSave(void) {/* code performing save command */}
int doCmdQuit(void) {/* code performing quit command */}
```

An example of this technique being used in real code is for GPU command dispatching in Chromium.

Chapter 36: Aliasing and effective type

Section 36.1: Effective type

The *effective type* of a data object is the last type information that was associated with it, if any.

```
// a normal variable, effective type uint32_t, and this type never changes
uint32_t a = 0.0;

// effective type of *pa is uint32_t, too, simply
// because *pa is the object a
uint32_t* pa = &a;

// the object pointed to by q has no effective type, yet
void* q = malloc(sizeof uint32_t);
// the object pointed to by q still has no effective type,
// because nobody has written to it
uint32_t* qb = q;
// *qb now has effective type uint32_t because a uint32_t value was written
*qb = 37;

// the object pointed to by r has no effective type, yet, although
// it is initialized
void* r = calloc(1, sizeof uint32_t);
// the object pointed to by r still has no effective type,
// because nobody has written to or read from it
uint32_t* rc = r;
// *rc now has effective type uint32_t because a value is read
// from it with that type. The read operation is valid because we used calloc.
// Now the object pointed to by r (which is the same as *rc) has
// gained an effective type, although we didn't change its value.
uint32_t c = *rc;

// the object pointed to by s has no effective type, yet.
void* s = malloc(sizeof uint32_t);
// the object pointed to by s now has effective type uint32_t
// because an uint32_t value is copied into it.
memcpy(s, r, sizeof uint32_t);
```

Observe that for the latter, it was not necessary that we even have an `uint32_t*` pointer to that object. The fact that we have copied another `uint32_t` object is sufficient.

Section 36.2: restrict qualification

If we have two pointer arguments of the same type, the compiler can't make any assumption and will always have to assume that the change to `*e` may change `*f`:

```
void fun(float* e, float* f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}

float fval = 4;
float eval = 77;
 fun(&eval, &fval);
```

all goes well and something like

> is 4 equal to 4?

is printed. If we pass the same pointer, the program will still do the right thing and print

> is 4 equal to 22?

This can turn out to be inefficient, if we *know* by some outside information that `e` and `f` will never point to the same data object. We can reflect that knowledge by adding `restrict` qualifiers to the pointer parameters:

```
void fan(float*restrict e, float*restrict f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}
```

Then the compiler may always suppose that `e` and `f` point to different objects.

Section 36.3: Changing bytes

Once an object has an effective type, you should not attempt to modify it through a pointer of another type, unless that other type is a character type, `char`, `signed char` or `unsigned char`.

```
#include <inttypes.h>
#include <stdio.h>

int main(void) {
  uint32_t a = 57;
  // conversion from incompatible types needs a cast !
  unsigned char* ap = (unsigned char*)&a;
  for (size_t i = 0; i < sizeof a; ++i) {
    /* set each byte of a to 42 */
    ap[i] = 42;
  }
  printf("a now has value %" PRIu32 "\n", a);
}
```

This is a valid program that prints

> a now has value 707406378

This works because:

- The access is made to the individual bytes seen with type `unsigned char` so each modification is well defined.
- The two views to the object, through `a` and through `*ap`, alias, but since `ap` is a pointer to a character type, the strict aliasing rule does not apply. Thus the compiler has to assume that the value of `a` may have been changed in the `for` loop. The modified value of `a` must be constructed from the bytes that have been changed.
- The type of `a`, `uint32_t` has no padding bits. All its bits of the representation count for the value, here

707406378, and there can be no trap representation.

Section 36.4: Character types cannot be accessed through non-character types

If an object is defined with static, thread, or automatic storage duration, and it has a character type, either: `char`, `unsigned char`, or `signed char`, it may not be accessed by a non-character type. In the below example a `char` array is reinterpreted as the type `int`, and the behavior is undefined on every dereference of the `int` pointer `b`.

```
int main( void )
{
    char a[100];
    int* b = ( int* )&a;
    *b = 1;

    static char c[100];
    b = ( int* )&c;
    *b = 2;

    _Thread_local char d[100];
    b = ( int* )&d;
    *b = 3;
}
```

This is undefined because it violates the "effective type" rule, no data object that has an effective type may be accessed through another type that is not a character type. Since the other type here is `int`, this is not allowed.

Even if alignment and pointer sizes would be known to fit, this would not exempt from this rule, behavior would still be undefined.

This means in particular that there is no way in standard C to reserve a buffer object of character type that can be used through pointers with different types, as you would use a buffer that was received by `malloc` or similar function.

A correct way to achieve the same goal as in the above example would be to use a `union`.

```
typedef union bufType bufType;
union bufType {
   char c[sizeof(int[25])];
   int i[25];
};

int main( void )
{
    bufType a = { .c = { 0 } }; // reserve a buffer and initialize
    int* b = a.i;      // no cast necessary
    *b = 1;

    static bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 2;

    _Thread_local bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 3;
}
```

Here, the `union` ensures that the compiler knows from the start that the buffer could be accessed through different

views. This also has the advantage that now the buffer has a "view" `a.i` that already is of type `int` and no pointer conversion is needed.

Section 36.5: Violating the strict aliasing rules

In the following code let us assume for simplicity that `float` and `uint32_t` have the same size.

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    float b = *f;
    print("%g should equal %g\n", a, b);
}
```

`u` and `f` have different base type, and thus the compiler can assume that they point to different objects. There is no possibility that `*f` could have changed between the two initializations of `a` and `b`, and so the compiler may optimize the code to something equivalent to

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    print("%g should equal %g\n", a, a);
}
```

That is, the second load operation of `*f` can be optimized out completely.

If we call this function "normally"

```
float fval = 4;
uint32_t uval = 77;
fun(&uval, &fval);
```

all goes well and something like

> 4 should equal 4

is printed. But if we cheat and pass the same pointer, after converting it,

```
float fval = 4;
uint32_t* up = (uint32_t*)&fval;
fun(up, &fval);
```

we violate the strict aliasing rule. Then the behavior becomes undefined. The output could be as above, if the compiler had optimized the second access, or something completely different, and so your program ends up in a completely unreliable state.

Chapter 37: Compilation

The C language is traditionally a compiled language (as opposed to interpreted). The C Standard defines ***translation phases***, and the product of applying them is a program image (or compiled program). In c11, the phases are listed in §5.1.1.2.

Section 37.1: The Compiler

After the C pre-processor has included all the header files and expanded all macros, the compiler can compile the program. It does this by turning the C source code into an object code file, which is a file ending in `.o` which contains the binary version of the source code. Object code is not directly executable, though. In order to make an executable, you also have to add code for all of the library functions that were `#include`d into the file (this is not the same as including the declarations, which is what `#include` does). This is the job of the linker.

In general, the exact sequence how to invoke a C compiler depends much on the system that you are using. Here we are using the GCC compiler, though it should be noted that many more compilers exist:

```
% gcc -Wall -c foo.c
```

`%` is the OS' command prompt. This tells the compiler to run the pre-processor on the file `foo.c` and then compile it into the object code file `foo.o`. The `-c` option means to compile the source code file into an object file but not to invoke the linker. This option `-c` is available on POSIX systems, such as Linux or macOS; other systems may use different syntax.

If your entire program is in one source code file, you can instead do this:

```
% gcc -Wall foo.c -o foo
```

This tells the compiler to run the pre-processor on `foo.c`, compile it and then link it to create an executable called `foo`. The `-o` option states that the next word on the line is the name of the binary executable file (program). If you don't specify the `-o`, (if you just type `gcc foo.c`), the executable will be named `a.out` for historical reasons.

In general the compiler takes four steps when converting a `.c` file into an executable:

1. **pre-processing** - textually expands `#include` directives and `#define` macros in your `.c` file
2. **compilation** - converts the program into assembly (you can stop the compiler at this step by adding the `-S` option)
3. **assembly** - converts the assembly into machine code
4. **linkage** - links the object code to external libraries to create an executable

Note also that the name of the compiler we are using is GCC, which stands for both "GNU C compiler" and "GNU compiler collection", depending on context. Other C compilers exist. For Unix-like operating systems, many of them have the name `cc`, for "C compiler", which is often a symbolic link to some other compiler. On Linux systems, `cc` is often an alias for GCC. On macOS or OS-X, it points to clang.

The POSIX standards currently mandates c99 as the name of a C compiler — it supports the C99 standard by default. Earlier versions of POSIX mandated c89 as the compiler. POSIX also mandates that this compiler understands the options `-c` and `-o` that we used above.

Note: The `-Wall` option present in both `gcc` examples tells the compiler to print warnings about questionable constructions, which is strongly recommended. It is a also good idea to add other warning options, e.g. `-Wextra`.

Section 37.2: File Types

Compiling C programs requires you to work with five kinds of files:

1. **Source files**: These files contain function definitions, and have names which end in `.c` by convention. Note: `.cc` and `.cpp` are C++ files; *not* C files.
   e.g., `foo.c`
2. **Header files**: These files contain function prototypes and various pre-processor statements (see below). They are used to allow source code files to access externally-defined functions. Header files end in `.h` by convention.
   e.g., `foo.h`
3. **Object files**: These files are produced as the output of the compiler. They consist of function definitions in binary form, but they are not executable by themselves. Object files end in `.o` by convention, although on some operating systems (e.g. Windows, MS-DOS), they often end in `.obj`.
   e.g., `foo.o` `foo.obj`
4. **Binary executables**: These are produced as the output of a program called a "linker". The linker links together a number of object files to produce a binary file which can be directly executed. Binary executables have no special suffix on Unix operating systems, although they generally end in `.exe` on Windows.
   e.g., `foo` `foo.exe`
5. **Libraries**: A library is a compiled binary but is not in itself an an executable (i.e., there is no `main()` function in a library). A library contains functions that may be used by more than one program. A library should ship with header files which contain prototypes for all functions in the library; these header files should be referenced (e.g; `#include <library.h>`) in any source file that uses the library. The linker then needs to be referred to the library so the program can successfully compiled. There are two types of libraries: static and dynamic.

   - **Static library**: A static library (`.a` files for POSIX systems and `.lib` files for Windows — not to be confused with [DLL import library files](), which also use the `.lib` extension) is statically built into the program . Static libraries have the advantage that the program knows exactly which version of a library is used. On the other hand, the sizes of executables are bigger as all used library functions are included.
     e.g., `libfoo.a` `foo.lib`
   - **Dynamic library**: A dynamic library (`.so` files for most POSIX systems, `.dylib` for OSX and `.dll` files for Windows) is dynamically linked at runtime by the program. These are also sometimes referred to as shared libraries because one library image can be shared by many programs. Dynamic libraries have the advantage of taking up less disk space if more than one application is using the library. Also, they allow library updates (bug fixes) without having to rebuild executables.
     e.g., `foo.so` `foo.dylib` `foo.dll`

Section 37.3: The Linker

The job of the linker is to link together a bunch of object files (`.o` files) into a binary executable. The process of *linking* mainly involves *resolving symbolic addresses to numerical addresses*. The result of the link process is normally an executable program.

During the link process, the linker will pick up all the object modules specified on the command line, add some system-specific *startup code* in front and try to resolve all *external* references in the object module with *external definitions* in other object files (object files can be specified directly on the command line or may implicitly be added through libraries). It will then assign *load addresses* for the object files, that is, it specifies where the code and data

will end up in the address space of the finished program. Once it's got the load addresses, it can replace all the symbolic addresses in the object code with "real", numerical addresses in the target's address space. The program is ready to be executed now.

This includes both the object files that the compiler created from your source code files as well as object files that have been pre-compiled for you and collected into library files. These files have names which end in `.a` or `.so`, and you normally don't need to know about them, as the linker knows where most of them are located and will link them in automatically as needed.

Implicit invocation of the linker

Like the pre-processor, the linker is a separate program, often called `ld` (but Linux uses `collect2`, for example). Also like the pre-processor, the linker is invoked automatically for you when you use the compiler. Thus, the normal way of using the linker is as follows:

```
% gcc foo.o bar.o baz.o -o myprog
```

This line tells the compiler to link together three object files (`foo.o`, `bar.o`, and `baz.o`) into a binary executable file named `myprog`. Now you have a file called `myprog` that you can run and which will hopefully do something cool and/or useful.

Explicit invocation of the linker

It is possible to invoke the linker directly, but this is seldom advisable, and is typically very platform-specific. That is, options that work on Linux won't necessarily work on Solaris, AIX, macOS, Windows, and similarly for any other platform. If you work with GCC, you can use `gcc -v` to see what is executed on your behalf.

Options for the linker

The linker also takes some arguments to modify it's behavior. The following command would tell gcc to link `foo.o` and `bar.o`, but also include the `ncurses` library.

```
% gcc foo.o bar.o -o foo -lncurses
```

This is actually (more or less) equivalent to

```
% gcc foo.o bar.o /usr/lib/libncurses.so -o foo
```

(although `libncurses.so` could be `libncurses.a`, which is just an archive created with `ar`). Note that you should list the libraries (either by pathname or via `-lname` options) after the object files. With static libraries, the order that they are specified matters; often, with shared libraries, the order doesn't matter.

Note that on many systems, if you are using mathematical functions (from **`<math.h>`**), you need to specify `-lm` to load the mathematics library — but Mac OS X and macOS Sierra do not require this. There are other libraries that are separate libraries on Linux and other Unix systems, but not on macOS — POSIX threads, and POSIX realtime, and networking libraries are examples. Consequently, the linking process varies between platforms.

Other compilation options

This is all you need to know to begin compiling your own C programs. Generally, we also recommend that you use the `-Wall` command-line option:

```
% gcc -Wall -c foo.cc
```

The `-Wall` option causes the compiler to warn you about legal but dubious code constructs, and will help you catch a lot of bugs very early.

If you want the compiler to throw more warnings at you (including variables that are declared but not used, forgetting to return a value etc.), you can use this set of options, as `-Wall`, despite the name, doesn't turn *all of the possible warnings* on:

```
% gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \
>     -Wmissing-declarations -Wredundant-decls -Wshadow …
```

Note that `clang` has an option [`-Weverything`] which really does turn on all warnings in `clang`.

Section 37.4: The Preprocessor

Before the C compiler starts compiling a source code file, the file is processed in a preprocessing phase. This phase can be done by a separate program or be completely integrated in one executable. In any case, it is invoked automatically by the compiler before compilation proper begins. The preprocessing phase converts your source code into another source code or translation unit by applying textual replacements. You can think of it as a "modified" or "expanded" source code. That expanded source may exist as a real file in the file system, or it may only be stored in memory for a short time before being processed further.

Preprocessor commands start with the pound sign ("#"). There are several preprocessor commands; two of the most important are:

1. **Defines**:

   `#define` is mainly used to define constants. For instance,

   ```
   #define BIGNUM 1000000
   int a = BIGNUM;
   ```

   becomes

   ```
   int a = 1000000;
   ```

   `#define` is used in this way so as to avoid having to explicitly write out some constant value in many different places in a source code file. This is important in case you need to change the constant value later on; it's much less bug-prone to change it once, in the `#define`, than to have to change it in multiple places scattered all over the code.

   Because `#define` just does advanced search and replace, you can also declare macros. For instance:

   ```
   #define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)
   // in the function:
   a = x;
   ISTRUE(a);
   ```

   becomes:

   ```
   // in the function:
   a = x;
   do {
       a = a ? 1 : 0;
   } while(0);
   ```

At first approximation, this effect is roughly the same as with inline functions, but the preprocessor doesn't provide type checking for `#define` macros. This is well known to be error-prone and their use necessitates great caution.

Also note here, that the preprocessor would also replace comments with a blanks as explained below.

2. **Includes**:

   `#include` is used to access function definitions defined outside of a source code file. For instance:

   ```
   #include <stdio.h>
   ```

   causes the preprocessor to paste the contents of **`<stdio.h>`** into the source code file at the location of the `#include` statement before it gets compiled. `#include` is almost always used to include header files, which are files which mainly contain function declarations and `#define` statements. In this case, we use `#include` in order to be able to use functions such as `printf` and `scanf`, whose declarations are located in the file `stdio.h`. C compilers do not allow you to use a function unless it has previously been declared or defined in that file; `#include` statements are thus the way to re-use previously-written code in your C programs.

3. **Logic operations**:

   ```
   #if defined A || defined B
   variable = another_variable + 1;
   #else
   variable = another_variable * 2;
   #endif
   ```

   will be changed to:

   ```
   variable = another_variable + 1;
   ```

   if A or B were defined somewhere in the project before. If this is not the case, of course the preprocessor will do this:

   ```
   variable = another_variable * 2;
   ```

   This is often used for code, that runs on different systems or compiles on different compilers. Since there are global defines, that are compiler/system specific you can test on those defines and always let the compiler just use the code he will compile for sure.

4. **Comments**

   The Preprocessor replaces all comments in the source file by single spaces. Comments are indicated by `//` up to the end of the line, or a combination of opening `/*` and closing `*/` comment brackets.

Section 37.5: The Translation Phases

As of the C 2011 Standard, listed in §*5.1.1.2 Translation Phases*, the translation of source code to program image (e.g., the executable) are listed to occur in 8 ordered steps.

1. The source file input is mapped to the source character set (if necessary). Trigraphs are replaced in this step.
2. Continuation lines (lines that end with `\`) are spliced with the next line.
3. The source code is parsed into whitespace and preprocessing tokens.
4. The preprocessor is applied, which executes directives, expands macros, and applies pragmas. Each source file pulled in by `#include` undergoes translation phases 1 through 4 (recursively if necessary). All preprocessor related directives are then deleted.
5. Source character set values in character constants and string literals are mapped to the execution character set.
6. String literals adjacent to each other are concatenated.
7. The source code is parsed into tokens, which comprise the translation unit.
8. External references are resolved, and the program image is formed.

An implementation of a C compiler may combine several steps together, but the resulting image must still behave as if the above steps had occurred separately in the order listed above.

Chapter 38: Inline assembly

Section 38.1: gcc Inline assembly in macros

We can put assembly instructions inside a macro and use the macro like you would call a function.

```
#define mov(x,y) \
{ \
    __asm__ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \
}

/// some definition and assignment
unsigned char sbox[size][size];
unsigned char sbox[size][size];

///Using
mov(state[0][1], sbox[si][sj]);
```

Using inline assembly instructions embedded in C code can improve the run time of a program. This is very helpful in time critical situations like cryptographic algorithms such as AES. For example, for a simple shift operation that is needed in the AES algorithm, we can substitute a direct `Rotate Right` assembly instruction with C shift operator `>>`.

In an implementation of 'AES256', in 'AddRoundKey()' function we have some statements like this:

```
unsigned int w;          // 32-bit
unsigned char subkey[4]; // 8-bit, 4*8 = 32

subkey[0] = w >> 24;     // hold 8 bit, MSB, leftmost group of 8-bits
subkey[1] = w >> 16;     // hold 8 bit, second group of 8-bit from left
subkey[2] = w >> 8;      // hold 8 bit, second group of 8-bit from right
subkey[3] = w;           // hold 8 bit, LSB, rightmost group of 8-bits

/// subkey <- w
```

They simply assign the bit value of `w` to `subkey` array.

We can change three shift + assign and one assign C expression with only one assembly `Rotate Right` operation.

```
__asm__ ("l.ror  %0,%1,%2" : "=r" (* (unsigned int *) subkey)  : "r" (w), "r" (0x10));
```

The final result is exactly same.

Section 38.2: gcc Basic asm support

Basic assembly support with gcc has the following syntax:

```
asm [ volatile ] ( AssemblerInstructions )
```

where `AssemblerInstructions` is the direct assembly code for the given processor. The volatile keyword is optional and has no effect as gcc does not optimize code within a basic asm statement. `AssemblerInstructions` can contain multiple assembly instructions. A basic asm statement is used if you have an asm routine that must exist outside of a C function. The following example is from the GCC manual:

```
 /* Note that this code will not compile with -masm=intel */
 #define DebugBreak() asm("int $3")
```

In this example, you could then use `DebugBreak()` in other places in your code and it will execute the assembly instruction `int $3`. Note that even though gcc will not modify any code in a basic asm statement, the optimizer may still move consecutive asm statements around. If you have multiple assembly instructions that must occur in a specific order, include them in one asm statement.

Section 38.3: gcc Extended asm support

Extended asm support in gcc has the following syntax:

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ])

 asm [volatile] goto ( AssemblerTemplate
                       :
                       : InputOperands
                       : Clobbers
                       : GotoLabels)
```

where `AssemblerTemplate` is the template for the assembler instruction, `OutputOperands` are any C variables that can be modified by the assembly code, `InputOperands` are any C variables used as input parameters, `Clobbers` are a list or registers that are modified by the assembly code, and `GotoLabels` are any goto statement labels that may be used in the assembly code.

The extended format is used within C functions and is the more typical usage of inline assembly. Below is an example from the Linux kernel for byte swapping 16-bit and 32-bit numbers for an ARM processor:

```
/* From arch/arm/include/asm/swab.h in Linux kernel version 4.6.4 */
#if __LINUX_ARM_ARCH__ >= 6

static inline __attribute_const__ __u32 __arch_swahb32(__u32 x)
{
    __asm__ ("rev16 %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swahb32 __arch_swahb32
#define __arch_swab16(x) ((__u16)__arch_swahb32(x))

static inline __attribute_const__ __u32 __arch_swab32(__u32 x)
{
    __asm__ ("rev %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swab32 __arch_swab32

#endif
```

Each asm section uses the variable `x` as its input and output parameter. The C function then returns the manipulated result.

With the extended asm format, gcc may optimize the assembly instructions in an asm block following the same rules it uses for optimizing C code. If you want your asm section to remain untouched, use the `volatile` keyword for the asm section.

Chapter 39: Identifier Scope

Section 39.1: Function Prototype Scope

```
#include <stdio.h>

/* The parameter name, apple, has function prototype scope.  These names
   are not significant outside the prototype itself.  This is demonstrated
   below. */

int test_function(int apple);

int main(void)
{
    int orange = 5;

    orange = test_function(orange);
    printf("%d\r\n", orange); //orange = 6

    return 0;
}

int test_function(int fruit)
{
    fruit += 1;
    return fruit;
}
```

Note that you get puzzling error messages if you introduce a type name in a prototype:

```
int function(struct whatever *arg);

struct whatever
{
    int a;
    // ...
};

int function(struct whatever *arg)
{
    return arg->a;
}
```

With GCC 6.3.0, this code (source file `dc11.c`) produces:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c
dc11.c:1:25: error: 'struct whatever' declared inside parameter list will not be visible outside of
this definition or declaration [-Werror]
     int function(struct whatever *arg);
                         ^~~~~~~~
dc11.c:9:9: error: conflicting types for 'function'
     int function(struct whatever *arg)
         ^~~~~~~~
dc11.c:1:9: note: previous declaration of 'function' was here
     int function(struct whatever *arg);
         ^~~~~~~~
cc1: all warnings being treated as errors
$
```

Place the structure definition before the function declaration, or add `struct whatever;` as a line before the function declaration, and there is no problem. You should not introduce new type names in a function prototype because there's no way to use that type, and hence no way to define or use that function.

Section 39.2: Block Scope

An identifier has block scope if its corresponding declaration appears inside a block (parameter declaration in function definition apply). The scope ends at the end of the corresponding block.

No different entities with the same identifier can have the same scope, but scopes may overlap. In case of overlapping scopes the only visible one is the one declared in the innermost scope.

```
#include <stdio.h>

void test(int bar)                   // bar has scope test function block
{
    int foo = 5;                     // foo has scope test function block
    {
        int bar = 10;                // bar has scope inner block, this overlaps with previous
test:bar declaration, and it hides test:bar
        printf("%d %d\n", foo, bar); // 5 10
    }                                // end of scope for inner bar
    printf("%d %d\n", foo, bar);     // 5 5, here bar is test:bar
}                                    // end of scope for test:foo and test:bar

int main(void)
{
    int foo = 3;         // foo has scope main function block

    printf("%d\n", foo); // 3
    test(5);
    printf("%d\n", foo); // 3
    return 0;
}                        // end of scope for main:foo
```

Section 39.3: File Scope

```
#include <stdio.h>

/* The identifier, foo, is declared outside all blocks.
   It can be used anywhere after the declaration until the end of
   the translation unit. */
static int foo;

void test_function(void)
{
    foo += 2;
}

int main(void)
{
    foo = 1;

    test_function();
    printf("%d\r\n", foo); //foo = 3;

    return 0;
}
```

Section 39.4: Function scope

Function scope is the special scope for **labels**. This is due to their unusual property. A **label** is visible through the entire function it is defined and one can jump (using instruction `goto`*`label`*) to it from any point in the same function. While not useful, the following example illustrate the point:

```
#include <stdio.h>

int main(int argc,char *argv[]) {
    int a = 0;
    goto INSIDE;
  OUTSIDE:
    if (a!=0) {
        int i=0;
      INSIDE:
        printf("a=%d\n",a);
        goto OUTSIDE;
    }
}
```

`INSIDE` may seem defined *inside* the `if` block, as it is the case for `i` which scope is the block, but it is not. It is visible in the whole function as the instruction `goto INSIDE;` illustrates. Thus there can't be two labels with the same identifier in a single function.

A possible usage is the following pattern to realize correct complex cleanups of allocated ressources:

```
#include <stdlib.h>
#include <stdio.h>

void a_function(void) {
   double* a = malloc(sizeof(double[34]));
   if (!a) {
      fprintf(stderr,"can't allocate\n");
      return;                 /* No point in freeing a if it is null */
   }
   FILE* b = fopen("some_file","r");
   if (!b) {
      fprintf(stderr,"can't open\n");
      goto CLEANUP1;          /* Free a; no point in closing b */
   }
   /* do something reasonable */
   if (error) {
      fprintf(stderr,"something's wrong\n");
      goto CLEANUP2;       /* Free a and close b to prevent leaks */
   }
   /* do yet something else */
CLEANUP2:
   close(b);
CLEANUP1:
   free(a);
}
```

Labels such as `CLEANUP1` and `CLEANUP2` are special identifiers that behave differently from all other identifiers. They are visible from everywhere inside the function, even in places that are executed before the labeled statement, or even in places that could never be reached if none of the `goto` is executed. Labels are often written in lower-case rather than upper-case.

Chapter 40: Implicit and Explicit Conversions

Section 40.1: Integer Conversions in Function Calls

Given that the function has a proper prototype, integers are widened for calls to functions according to the rules of integer conversion, C11 6.3.1.3.

> **6.3.1.3 Signed and unsigned integers**
> When a value with integer type is converted to another integer type other than _Bool, if the value can be represented by the new type, it is unchanged.
>
> Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.
>
> Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

Usually you should not truncate a wide signed type to a narrower signed type, because obviously the values can't fit and there is no clear meaning that this should have. The C standard cited above defines these cases to be "implementation-defined", that is, they are not portable.

The following example supposes that `int` is 32 bit wide.

```
#include <stdio.h>
#include <stdint.h>

void param_u8(uint8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_u16(uint16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_u32(uint32_t val) {
    printf("%s val is %u\n", __func__, val);  /* here val fits into unsigned */
}

void param_u64(uint64_t val) {
    printf("%s val is " PRI64u "\n", __func__, val); /* Fixed with format string */
}

void param_s8(int8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s16(int16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s32(int32_t val) {
    printf("%s val is %d\n", __func__, val); /* val has same width as int */
}
```

```c
void param_s64(int64_t val) {
    printf("%s val is " PRI64d "\n", __func__, val); /* Fixed with format string */
}

int main(void) {

    /* Declare integers of various widths */
    uint8_t  u8  = 127;
    uint8_t  s64  = INT64_MAX;

    /* Integer argument is widened when function parameter is wider */
    param_u8(u8);   /* param_u8 val is 127 */
    param_u16(u8);  /* param_u16 val is 127 */
    param_u32(u8);  /* param_u32 val is 127 */
    param_u64(u8);  /* param_u64 val is 127 */
    param_s8(u8);   /* param_s8 val is 127 */
    param_s16(u8);  /* param_s16 val is 127 */
    param_s32(u8);  /* param_s32 val is 127 */
    param_s64(u8);  /* param_s64 val is 127 */

    /* Integer argument is truncated when function parameter is narrower */
    param_u8(s64);  /* param_u8 val is 255 */
    param_u16(s64); /* param_u16 val is 65535 */
    param_u32(s64); /* param_u32 val is 4294967295 */
    param_u64(s64); /* param_u64 val is 9223372036854775807 */
    param_s8(s64);  /* param_s8 val is implementation defined */
    param_s16(s64); /* param_s16 val is implementation defined */
    param_s32(s64); /* param_s32 val is implementation defined */
    param_s64(s64); /* param_s64 val is 9223372036854775807 */

    return 0;
}
```

Section 40.2: Pointer Conversions in Function Calls

Pointer conversions to `void*` are implicit, but any other pointer conversion must be explicit. While the compiler allows an explicit conversion from any pointer-to-data type to any other pointer-to-data type, accessing an object through a wrongly typed pointer is erroneous and leads to undefined behavior. The only case that these are allowed are if the types are compatible or if the pointer with which your are looking at the object is a character type.

```c
#include <stdio.h>

void func_voidp(void* voidp) {
    printf("%s Address of ptr is %p\n", __func__, voidp);
}

/* Structures have same shape, but not same type */
struct struct_a {
    int a;
    int b;
} data_a;

struct struct_b {
    int a;
    int b;
} data_b;

void func_struct_b(struct struct_b* bp) {
    printf("%s Address of ptr is %p\n", __func__, (void*) bp);
}
```

```c
int main(void) {


    /* Implicit ptr conversion allowed for void* */
    func_voidp(&data_a);

    /*
     * Explicit ptr conversion for other types
     *
     * Note that here although the have identical definitions,
     * the types are not compatible, and that the this call is
     * erroneous and leads to undefined behavior on execution.
     */
    func_struct_b((struct struct_b*)&data_a);

    /* My output shows: */
    /* func_charp Address of ptr is 0x601030 */
    /* func_voidp Address of ptr is 0x601030 */
    /* func_struct_b Address of ptr is 0x601030 */

    return 0;
}
```

Chapter 41: Type Qualifiers

Section 41.1: Volatile variables

The volatile keyword tells the compiler that the value of the variable may change at any time as a result of external conditions, not only as a result of program control flow.

The compiler will not optimize anything that has to do with the volatile variable.

```
volatile int foo; /* Different ways to declare a volatile variable */
int volatile foo;

volatile uint8_t * pReg; /* Pointers to volatile variable */
uint8_t volatile * pReg;
```

There are two main reasons to uses volatile variables:

- To interface with hardware that has memory-mapped I/O registers.
- When using variables that are modified outside the program control flow (e.g., in an interrupt service routine)

Let's see this example:

```
int quit = false;

void main()
{
    ...
    while (!quit) {
      // Do something that does not modify the quit variable
    }
    ...
}

void interrupt_handler(void)
{
  quit = true;
}
```

The compiler is allowed to notice the while loop does not modify the quit variable and convert the loop to a endless while (true) loop. Even if the quit variable is set on the signal handler for SIGINT and SIGTERM, the compiler does not know that.

Declaring quit as volatile will tell the compiler to not optimize the loop and the problem will be solved.

The same problem happens when accessing hardware, as we see in this example:

```
uint8_t * pReg = (uint8_t *) 0x1717;

// Wait for register to become non-zero
while (*pReg == 0) { } // Do something else
```

The behavior of the optimizer is to read the variable's value once, there is no need to reread it, since the value will always be the same. So we end up with an infinite loop. To force the compiler to do what we want, we modify the declaration to:

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1717;
```

Section 41.2: Unmodifiable (const) variables

```
const int a = 0; /* This variable is "unmodifiable", the compiler
                    should throw an error when this variable is changed */
int b = 0; /* This variable is modifiable */

b += 10; /* Changes the value of 'b' */
a += 10; /* Throws a compiler error */
```

The `const` qualification only means that we don't have the right to change the data. It doesn't mean that the value cannot change behind our back.

```
_Bool doIt(double const* a) {
   double rememberA = *a;
   // do something long and complicated that calls other functions

   return rememberA == *a;
}
```

During the execution of the other calls `*a` might have changed, and so this function may return either **`false`** or **`true`**.

Warning

Variables with `const` qualification could still be changed using pointers:

```
const int a = 0;

int *a_ptr = (int*)&a; /* This conversion must be explicitly done with a cast */
*a_ptr += 10;          /* This has undefined behavior */

printf("a = %d\n", a); /* May print: "a = 10" */
```

But doing so is an error that leads to undefined behavior. The difficulty here is that this may behave as expected in simple examples as this, but then go wrong when the code grows.

Chapter 42: Typedef

The `typedef` mechanism allows the creation of aliases for other types. It does not create new types. People often use `typedef` to improve the portability of code, to give aliases to structure or union types, or to create aliases for function (or function pointer) types.

In the C standard, `typedef` is classified as a 'storage class' for convenience; it occurs syntactically where storage classes such as `static` or **`extern`** could appear.

Section 42.1: Typedef for Structures and Unions

You can give alias names to a `struct`:

```
typedef struct Person {
    char name[32];
    int age;
} Person;

Person person;
```

Compared to the traditional way of declaring structs, programmers wouldn't need to have `struct` every time they declare an instance of that struct.

Note that the name `Person` (as opposed to `struct Person`) is not defined until the final semicolon. Thus for linked lists and tree structures which need to contain a pointer to the same structure type, you must use either:

```
typedef struct Person {
    char name[32];
    int age;
    struct Person *next;
} Person;
```

or:

```
typedef struct Person Person;

struct Person {
    char name[32];
    int age;
    Person *next;
};
```

The use of a `typedef` for a `union` type is very similar.

```
typedef union Float Float;

union Float
{
    float f;
    char  b[sizeof(float)];
};
```

A structure similar to this can be used to analyze the bytes that make up a `float` value.

Section 42.2: Typedef for Function Pointers

We can use typedef to simplify the usage of function pointers. Imagine we have some functions, all having the same signature, that use their argument to print out something in different ways:

```
#include<stdio.h>

void print_to_n(int n)
{
    for (int i = 1; i <= n; ++i)
        printf("%d\n", i);
}

void print_n(int n)
{
    printf("%d\n, n);
}
```

Now we can use a typedef to create a named function pointer type called printer:

```
typedef void (*printer_t)(int);
```

This creates a type, named printer_t for a pointer to a function that takes a single int argument and returns nothing, which matches the signature of the functions we have above. To use it we create a variable of the created type and assign it a pointer to one of the functions in question:

```
printer_t p = &print_to_n;
void (*p)(int) = &print_to_n; // This would be required without the type
```

Then to call the function pointed to by the function pointer variable:

```
p(5);           // Prints 1 2 3 4 5 on separate lines
(*p)(5);        // So does this
```

Thus the typedef allows a simpler syntax when dealing with function pointers. This becomes more apparent when function pointers are used in more complex situations, such as arguments to functions.

If you are using a function that takes a function pointer as a parameter without a function pointer type defined the function definition would be,

```
void foo (void (*printer)(int), int y){
    //code
    printer(y);
    //code
}
```

However, with the typedef it is:

```
void foo (printer_t printer, int y){
    //code
    printer(y);
    //code
}
```

Likewise functions can return function pointers and again, the use of a typedef can make the syntax simpler when doing so.

A classic example is the `signal` function from `<signal.h>`. The declaration for it (from the C standard) is:

```
void (*signal(int sig, void (*func)(int)))(int);
```

That's a function that takes two arguments — an `int` and a pointer to a function which takes an `int` as an argument and returns nothing — and which returns a pointer to function like its second argument.

If we defined a type `SigCatcher` as an alias for the pointer to function type:

```
typedef void (*SigCatcher)(int);
```

then we could declare `signal()` using:

```
SigCatcher signal(int sig, SigCatcher func);
```

On the whole, this is easier to understand (even though the C standard did not elect to define a type to do the job). The `signal` function takes two arguments, an `int` and a `SigCatcher`, and it returns a `SigCatcher` — where a `SigCatcher` is a pointer to a function that takes an `int` argument and returns nothing.

Although using `typedef` names for pointer to function types makes life easier, it can also lead to confusion for others who will maintain your code later on, so use with caution and proper documentation. See also Function Pointers.

Section 42.3: Simple Uses of Typedef

For giving short names to a data type

Instead of:

```
long long int foo;
struct mystructure object;
```

one can use

```
/* write once */
typedef long long ll;
typedef struct mystructure mystruct;

/* use whenever needed */
ll foo;
mystruct object;
```

This reduces the amount of typing needed if the type is used many times in the program.

Improving portability

The attributes of data types vary across different architectures. For example, an `int` may be a 2-byte type in one implementation and an 4-byte type in another. Suppose a program needs to use a 4-byte type to run correctly.

In one implementation, let the size of `int` be 2 bytes and that of `long` be 4 bytes. In another, let the size of `int` be 4 bytes and that of `long` be 8 bytes. If the program is written using the second implementation,

```
/* program expecting a 4 byte integer */
int foo; /* need to hold 4 bytes to work */
/* some code involving many more ints */
```

For the program to run in the first implementation, all the `int` declarations will have to be changed to `long`.

```
/* program now needs long */
long foo; /*need to hold 4 bytes to work */
/* some code involving many more longs - lot to be changed */
```

To avoid this, one can use `typedef`

```
/* program expecting a 4 byte integer */
typedef int myint; /* need to declare once - only one line to modify if needed */
myint foo; /* need to hold 4 bytes to work */
/* some code involving many more myints */
```

Then, only the `typedef` statement would need to be changed each time, instead of examining the whole program.

Version ≥ C99

The **`<stdint.h>`** header and the related **`<inttypes.h>`** header define standard type names (using `typedef`) for integers of various sizes, and these names are often the best choice in modern code that needs fixed size integers. For example, `uint8_t` is an unsigned 8-bit integer type; `int64_t` is a signed 64-bit integer type. The type `uintptr_t` is an unsigned integer type big enough to hold any pointer to object. These types are theoretically optional — but it is rare for them not to be available. There are variants like `uint_least16_t` (the smallest unsigned integer type with at least 16 bits) and `int_fast32_t` (the fastest signed integer type with at least 32 bits). Also, `intmax_t` and `uintmax_t` are the largest integer types supported by the implementation. These types are mandatory.

To specify a usage or to improve readability

If a set of data has a particular purpose, one can use `typedef` to give it a meaningful name. Moreover, if the property of the data changes such that the base type must change, only the `typedef` statement would have to be changed, instead of examining the whole program.

Chapter 43: Storage Classes

A storage class is used to set the scope of a variable or function. By knowing the storage class of a variable, we can determine the life-time of that variable during the run-time of the program.

Section 43.1: auto

This storage class denotes that an identifier has automatic storage duration. This means once the scope in which the identifier was defined ends, the object denoted by the identifier is no longer valid.

Since all objects, not living in global scope or being declared `static`, have automatic storage duration by default when defined, this keyword is mostly of historical interest and should not be used:

```
int foo(void)
{
    /* An integer with automatic storage duration. */
    auto int i = 3;

    /* Same */
    int j = 5;

    return 0;
} /* The values of i and j are no longer able to be used. */
```

Section 43.2: register

Hints to the compiler that access to an object should be as fast as possible. Whether the compiler actually uses the hint is implementation-defined; it may simply treat it as equivalent to `auto`.

The only property that is definitively different for all objects that are declared with `register` is that they cannot have their address computed. Thereby `register` can be a good tool to ensure certain optimizations:

```
register size_t size = 467;
```

is an object that can never *alias* because no code can pass its address to another function where it might be changed unexpectedly.

This property also implies that an array

```
register int array[5];
```

cannot decay into a pointer to its first element (i.e. `array` turning into `&array[0]`). This means that the elements of such an array cannot be accessed and the array itself cannot be passed to a function.

In fact, the only legal usage of an array declared with a `register` storage class is the `sizeof` operator; any other operator would require the address of the first element of the array. For that reason, arrays generally should not be declared with the `register` keyword since it makes them useless for anything other than size computation of the entire array, which can be done just as easily without the `register` keyword.

The `register` storage class is more appropriate for variables that are defined inside a block and are accessed with high frequency. For example,

```
/* prints the sum of the first 5 integers*/
/* code assumed to be part of a function body*/
```

```
{
    register int k, sum;
    for(k = 1, sum = 0; k < 6; sum += k, k++);
        printf("\t%d\n",sum);
}
```

Version ≥ C11

The `_Alignof` operator is also allowed to be used with `register` arrays.

Section 43.3: static

The `static` storage class serves different purposes, depending on the location of the declaration in the file:

1. To confine the identifier to that translation unit only (scope=file).

```
/* No other translation unit can use this variable. */
static int i;

/* Same; static is attached to the function type of f, not the return type int. */
static int f(int n);
```

2. To save data for use with the next call of a function (scope=block):

```
void foo()
{
    static int a = 0; /* has static storage duration and its lifetime is the
                       * entire execution of the program; initialized to 0 on
                       * first function call */
    int b = 0; /* b has block scope and has automatic storage duration and
                * only "exists" within function */

    a += 10;
    b += 10;

    printf("static int a = %d, int b = %d\n", a, b);
}

int main(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        foo();
    }

    return 0;
}
```

This code prints:

```
static int a = 10, int b = 10
static int a = 20, int b = 10
static int a = 30, int b = 10
static int a = 40, int b = 10
static int a = 50, int b = 10
```

Static variables retain their value even when called from multiple different threads.

Version ≥ C99

3. Used in function parameters to denote an array is expected to have a constant minimum number of elements and a non-null parameter:

```
/* a is expected to have at least 512 elements. */
void printInts(int a[static 512])
{
    size_t i;
    for (i = 0; i < 512; ++i)
        printf("%d\n", a[i]);
}
```

The required number of items (or even a non-null pointer) is not necessarily checked by the compiler, and compilers are not required to notify you in any way if you don't have enough elements. If a programmer passes fewer than 512 elements or a null pointer, undefined behavior is the result. Since it is impossible to enforce this, extra care must be used when passing a value for that parameter to such a function.

Section 43.4: typedef

Defines a new type based on an existing type. Its syntax mirrors that of a variable declaration.

```
/* Byte can be used wherever `unsigned char` is needed */
typedef unsigned char Byte;

/* Integer is the type used to declare an array consisting of a single int */
typedef int Integer[1];

/* NodeRef is a type used for pointers to a structure type with the tag "node" */
typedef struct node *NodeRef;

/* SigHandler is the function pointer type that gets passed to the signal function. */
typedef void (*SigHandler)(int);
```

While not technically a storage class, a compiler will treat it as one since none of the other storage classes are allowed if the `typedef` keyword is used.

The `typedef`s are important and should not be substituted with `#define` macro.

```
typedef int newType;
newType *ptr;        // ptr is pointer to variable of type 'newType' aka int
```

However,

```
#define int newType
newType *ptr;        // Even though macros are exact replacements to words, this doesn't result to
a pointer to variable of type 'newType' aka int
```

Section 43.5: extern

Used to **declare an object or function** that is defined elsewhere (and that has *external linkage*). In general, it is used to declare an object or function to be used in a module that is not the one in which the corresponding object or function is defined:

```c
/* file1.c */
int foo = 2;  /* Has external linkage since it is declared at file scope. */
```

```c
/* file2.c */
#include <stdio.h>
int main(void)
{
    /* `extern` keyword refers to external definition of `foo`. */
    extern int foo;
    printf("%d\n", foo);
    return 0;
}
```

Version ≥ C99

Things get slightly more interesting with the introduction of the **inline** keyword in C99:

```c
/* Should usually be place in a header file such that all users see the definition */
/* Hints to the compiler that the function `bar` might be inlined */
/* and suppresses the generation of an external symbol, unless stated otherwise. */
inline void bar(int drink)
{
    printf("You ordered drink no.%d\n", drink);
}

/* To be found in just one .c file.
   Creates an external function definition of `bar` for use by other files.
   The compiler is allowed to choose between the inline version and the external
   definition when `bar` is called. Without this line, `bar` would only be an inline
   function, and other files would not be able to call it. */
extern void bar(int);
```

Section 43.6: _Thread_local

Version ≥ C11

This was a new storage specifier introduced in C11 along with multi-threading. This isn't available in earlier C standards.

Denotes *thread storage duration*. A variable declared with `_Thread_local` storage specifier denotes that the object is *local to that thread* and its lifetime is the entire execution of the thread in which it's created. It can also appear along with `static` or **extern**.

```c
#include <threads.h>
#include <stdio.h>
#define SIZE 5

int thread_func(void *id)
{
    /* thread local variable i. */
    static _Thread_local int i;

    /* Prints the ID passed from main() and the address of the i.
     * Running this program will print different addresses for i, showing
     * that they are all distinct objects. */
    printf("From thread:[%d], Address of i (thread local): %p\n", *(int*)id, (void*)&i);

    return 0;
}

int main(void)
```

```
{
    thrd_t id[SIZE];
    int arr[SIZE] = {1, 2, 3, 4, 5};

    /* create 5 threads. */
    for(int i = 0; i < SIZE; i++) {
        thrd_create(&id[i], thread_func, &arr[i]);
    }

    /* wait for threads to complete. */
    for(int i = 0; i < SIZE; i++) {
        thrd_join(id[i], NULL);
    }
}
```

Chapter 44: Declarations

Section 44.1: Calling a function from another C file

foo.h

```
#ifndef FOO_DOT_H    /* This is an "include guard" */
#define FOO_DOT_H    /* prevents the file from being included twice. */
                     /* Including a header file twice causes all kinds */
                     /* of interesting problems.*/

/**
 * This is a function declaration.
 * It tells the compiler that the function exists somewhere.
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

foo.c

```
#include "foo.h"    /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.  Put this
                     * header first in foo.c to ensure the header is self-contained.
                     */
#include <stdio.h>

/**
 * This is the function definition.
 * It is the actual body of the function which was declared elsewhere.
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");\n", id, name);
    /* This will print how foo was called to stderr - standard error.
     * e.g., foo(42, "Hi!") will print `foo(42, "Hi!")`
     */
}
```

main.c

```
#include "foo.h"

int main(void)
{
    foo(42, "bar");
    return 0;
}
```

Compile and Link

First, we *compile* both `foo.c` and `main.c` to *object files*. Here we use the `gcc` compiler, your compiler may have a different name and need other options.

```
$ gcc -Wall -c foo.c
$ gcc -Wall -c main.c
```

Now we link them together to produce our final executable:

```
$ gcc -o testprogram foo.o main.o
```

Section 44.2: Using a Global Variable

Use of global variables is generally discouraged. It makes your program more difficult to understand, and harder to debug. But sometimes using a global variable is acceptable.

global.h

```
#ifndef GLOBAL_DOT_H    /* This is an "include guard" */
#define GLOBAL_DOT_H

/**
 * This tells the compiler that g_myglobal exists somewhere.
 * Without "extern", this would create a new variable named
 * g_myglobal in _every file_ that included it. Don't miss this!
 */
extern int g_myglobal; /* _Declare_ g_myglobal, that is promise it will be _defined_ by
                        * some module. */

#endif /* GLOBAL_DOT_H */
```

global.c

```
#include "global.h" /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.
                     */

int g_myglobal;     /* _Define_ my_global. As living in global scope it gets initialised to 0
                     * on program start-up. */
```

main.c

```
#include "global.h"

int main(void)
{
    g_myglobal = 42;
    return 0;
}
```

See also How do I use **`extern`** to share variables between source files?

Section 44.3: Introduction

Example of declarations are:

```
int a; /* declaring single identifier of type int */
```

The above declaration declares single identifier named a which refers to some object with `int` type.

```
int a1, b1; /* declaring 2 identifiers of type int */
```

The second declaration declares 2 identifiers named `a1` and `b1` which refers to some other objects though with the same `int` type.

Basically, the way this works is like this - first you put some type, then you write a single or multiple expressions separated via comma (`,`) (**which will not be evaluated at this point - and which should otherwise be referred to as declarators in this context**). In writing such expressions, you are allowed to apply only the indirection (`*`), function call (`( )`) or subscript (or array indexing - `[ ]`) operators onto some identifier (you can also not use any operators at all). The identifier used is not required to be visible in the current scope. Some examples:

```
/* 1 */ int /* 2 */ (*z) /* 3 */ , /* 4 */ *x , /* 5 */ **c /* 6 */ ;
```

#	Description
1	The name of integer type.
2	Un-evaluated expression applying indirection to some identifier `z`.
3	We have a comma indicating that one more expression will follow in the same declaration.
4	Un-evaluated expression applying indirection to some other identifier `x`.
5	Un-evaluated expression applying indirection to the value of the expression `(*c)`.
6	End of declaration.

Note that none of the above identifiers were visible prior to this declaration and so the expressions used would not be valid before it.

After each such expression, the identifier used in it is introduced into the current scope. (If the identifier has assigned linkage to it, it may also be re-declared with the same type of linkage so that both identifiers refer to the same object or function)

Additionally, the equal operator sign (`=`) may be used for initialization. If an unevaluated expression (declarator) is followed by `=` inside the declaration - we say that the identifier being introduced is also being initialized. After the `=` sign we can put once again some expression, but this time it'll be evaluated and its value will be used as initial for the object declared.

Examples:

```
int l = 90; /* the same as: */

int l; l = 90; /* if it the declaration of l was in block scope */

int c = 2, b[c]; /* ok, equivalent to: */

int c = 2; int b[c];
```

Later in your code, you are allowed to write the exact same expression from the declaration part of the newly introduced identifier, giving you an object of the type specified at the beginning of the declaration, assuming that you've assigned valid values to all accessed objects in the way. Examples:

```
void f()
{
    int b2; /* you should be able to write later in your code b2
            which will directly refer to the integer object
            that b2 identifies */

    b2 = 2; /* assign a value to b2 */

    printf("%d", b2); /*ok - should print 2*/

    int *b3; /* you should be able to write later in your code *b3 */
```

```c
    b3 = &b2; /* assign valid pointer value to b3 */

    printf("%d", *b3); /* ok - should print 2 */

    int **b4; /* you should be able to write later in your code **b4 */

    b4 = &b3;

    printf("%d", **b4); /* ok - should print 2 */

    void (*p)(); /* you should be able to write later in your code (*p)() */

    p = &f; /* assign a valid pointer value */

    (*p)(); /* ok - calls function f by retrieving the
            pointer value inside p -    p
            and dereferencing it -      *p
            resulting in a function
            which is then called -      (*p)() -

            it is not *p() because else first the () operator is
            applied to p and then the resulting void object is
            dereferenced which is not what we want here */
}
```

The declaration of `b3` specifies that you can potentially use `b3` value as a mean to access some integer object.

Of course, in order to apply indirection (`*`) to `b3`, you should also have a proper value stored in it (see pointers for more info). You should also first store some value into an object before trying to retrieve it (you can see more about this problem here). We've done all of this in the above examples.

```c
int a3(); /* you should be able to call a3 */
```

This one tells the compiler that you'll attempt to call `a3`. In this case `a3` refers to function instead of an object. One difference between object and function is that functions will always have some sort of linkage. Examples:

```c
void f1()
{
    {
        int f2(); /* 1 refers to some function f2 */
    }

    {
        int f2(); /* refers to the exact same function f2 as (1) */
    }
}
```

In the above example, the 2 declarations refer to the same function `f2`, whilst if they were declaring objects then in this context (having 2 different block scopes), they would have be 2 different distinct objects.

```c
int (*a3)(); /* you should be able to apply indirection to `a3` and then call it */
```

Now it may seems to be getting complicated, but if you know operators precedence you'll have 0 problems reading the above declaration. The parentheses are needed because the `*` operator has less precedence then the `( )` one.

In the case of using the subscript operator, the resulting expression wouldn't be actually valid after the declaration because the index used in it (the value inside `[` and `]`) will always be 1 above the maximum allowed value for this object/function.

```
int a4[5]; /* here a4 shouldn't be accessed using the index 5 later on */
```

But it should be accessible by all other indexes lower then 5. Examples:

```
a4[0], a4[1]; a4[4];
```

`a4[5]` will result into UB. More information about arrays can be found here.

```
int (*a5)[5](); /* here a4 could be applied indirection
                indexed up to (but not including) 5
                and called */
```

Unfortunately for us, although syntactically possible, the declaration of `a5` is forbidden by the current standard.

Section 44.4: Typedef

Typedefs are declarations which have the keyword `typedef` in front and before the type. E.g.:

```
typedef int (*(*t0)())[5];
```

(*you can technically put the typedef after the type too - like this* `int typedef (*(*t0)())[5];` *but this is discouraged*)

The above declarations declares an identifier for a typedef name. You can use it like this afterwards:

```
t0 pf;
```

Which will have the same effect as writing:

```
int (*(*pf)())[5];
```

As you can see the typedef name "saves" the declaration as a type to use later for other declarations. This way you can save some keystrokes. Also as declaration using `typedef` is still a declaration you are not limited only by the above example:

```
t0 (*pf1);
```

Is the same as:

```
int (*(**pf1)())[5];
```

Section 44.5: Using Global Constants

Headers may be used to declare globally used read-only resources, like string tables for example.

Declare those in a separate header which gets included by any file ("*Translation Unit*") which wants to make use of them. It's handy to use the same header to declare a related enumeration to identify all string-resources:

resources.h:

```
#ifndef RESOURCES_H
#define RESOURCES_H

typedef enum { /* Define a type describing the possible valid resource IDs. */
  RESOURCE_UNDEFINED = -1, /* To be used to initialise any EnumResourceID typed variable to be
```

```
                              marked as "not in use", "not in list", "undefined", wtf.
                              Will say un-initialised on application level, not on language level.
Initialised uninitialised, so to say ;-)
                              Its like NULL for pointers ;-)*/
  RESOURCE_UNKNOWN = 0,     /* To be used if the application uses some resource ID,
                              for which we do not have a table entry defined, a fall back in
                              case we _need_ to display something, but do not find anything
                              appropriate. */

  /* The following identify the resources we have defined: */
  RESOURCE_OK,
  RESOURCE_CANCEL,
  RESOURCE_ABORT,
  /* Insert more here. */

  RESOURCE_MAX /* The maximum number of resources defined. */
} EnumResourceID;


extern const char * const resources[RESOURCE_MAX]; /* Declare, promise to anybody who includes
                                      this, that at linkage-time this symbol will be around.
                                      The 1st const guarantees the strings will not change,
                                      the 2nd const guarantees the string-table entries
                                      will never suddenly point somewhere else as set during
                                      initialisation. */
#endif
```

To actually define the resources created a related .c-file, that is another translation unit holding the actual instances of the what had been declared in the related header (.h) file:

resources.c:

```
#include "resources.h" /* To make sure clashes between declaration and definition are
                          recognised by the compiler include the declaring header into
                          the implementing, defining translation unit (.c file).

/* Define the resources. Keep the promise made in resources.h. */
const char * const resources[RESOURCE_MAX] = {
  "<unknown>",
  "OK",
  "Cancel",
  "Abort"
};
```

A program using this could look like this:

main.c:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>

#include "resources.h"


int main(void)
{
  EnumResourceID resource_id = RESOURCE_UNDEFINED;

  while ((++resource_id) < RESOURCE_MAX)
  {
```

```
    printf("resource ID: %d, resource: '%s'\n", resource_id, resources[resource_id]);
  }

  return EXIT_SUCCESS;
}
```

Compile the three file above using GCC, and link them to become the program file `main` for example using this:

```
gcc -Wall -Wextra -pedantic -Wconversion -g  main.c resources.c -o main
```

(use these `-Wall -Wextra -pedantic -Wconversion` to make the compiler really picky, so you don't miss anything before posting the code to SO, will say the world, or even worth deploying it into production)

Run the program created:

```
$ ./main
```

And get:

```
resource ID: 0, resource: ''
resource ID: 1, resource: 'OK'
resource ID: 2, resource: 'Cancel'
resource ID: 3, resource: 'Abort'
```

Section 44.6: Using the right-left or spiral rule to decipher C declaration

The "right-left" rule is a completely regular rule for deciphering C declarations. It can also be useful in creating them.

Read the symbols as you encounter them in the declaration...

```
*   as "pointer to"          - always on the left side
[]  as "array of"            - always on the right side
()  as "function returning"  - always on the right side
```

How to apply the rule

STEP 1

Find the identifier. This is your starting point. Then say to yourself, "identifier is." You've started your declaration.

STEP 2

Look at the symbols on the right of the identifier. If, say, you find `()` there, then you know that this is the declaration for a function. So you would then have *"identifier is function returning"*. Or if you found a `[]` there, you would say *"identifier is array of"*. Continue right until you run out of symbols OR hit a right parenthesis `)`. (If you hit a left parenthesis `(`, that's the beginning of a `()` symbol, even if there is stuff in between the parentheses. More on that below.)

STEP 3

Look at the symbols to the left of the identifier. If it is not one of our symbols above (say, something like "int"), just say it. Otherwise, translate it into English using that table above. Keep going left until you run out of symbols OR hit

a left parenthesis `(`.

Now repeat steps 2 and 3 until you've formed your declaration.

Here are some examples:

```
int *p[];
```

First, find identifier:

```
int *p[];
     ^
```

"p is"

Now, move right until out of symbols or right parenthesis hit.

```
int *p[];
      ^^
```

"p is array of"

Can't move right anymore (out of symbols), so move left and find:

```
int *p[];
    ^
```

"p is array of pointer to"

Keep going left and find:

```
int *p[];
^^^
```

"p is array of pointer to int".

(or *"p is an array where each element is of type pointer to int"*)

Another example:

```
int *(*func())();
```

Find the identifier.

```
int *(*func())();
       ^^^^
```

"func is"

Move right.

```
int *(*func())();
           ^^
```

"func is function returning"

Can't move right anymore because of the right parenthesis, so move left.

```
int *(*func())();
      ^
```

"func is function returning pointer to"

Can't move left anymore because of the left parenthesis, so keep going right.

```
int *(*func())();
              ^^
```

"func is function returning pointer to function returning"

Can't move right anymore because we're out of symbols, so go left.

```
int *(*func())();
    ^
```

"func is function returning pointer to function returning pointer to"

And finally, keep going left, because there's nothing left on the right.

```
int *(*func())();
^^^
```

"func is function returning pointer to function returning pointer to int".

As you can see, this rule can be quite useful. You can also use it to sanity check yourself while you are creating declarations, and to give you a hint about where to put the next symbol and whether parentheses are required.

Some declarations look much more complicated than they are due to array sizes and argument lists in prototype form. If you see `[3]`, that's read as *"array (size 3) of..."*. If you see `(char *,int)` that's read as *"function expecting (char ,int) and returning..."*.

Here's a fun one:

```
int (*(*fun_one)(char *,double))[9][20];
```

I won't go through each of the steps to decipher this one.

"fun_one is pointer to function expecting (char ,double) and returning pointer to array (size 9) of array (size 20) of int."

As you can see, it's not as complicated if you get rid of the array sizes and argument lists:

```
int (*(*fun_one)())[][];
```

You can decipher it that way, and then put in the array sizes and argument lists later.

Some final words:

It is quite possible to make illegal declarations using this rule, so some knowledge of what's legal in C is necessary. For instance, if the above had been:

```
int *((*fun_one)())[][];
```

it would have read *"fun_one is pointer to function returning array of array of pointer to int"*. Since a function cannot return an array, but only a pointer to an array, that declaration is illegal.

Illegal combinations include:

```
[]() - cannot have an array of functions
()() - cannot have a function that returns a function
()[] - cannot have a function that returns an array
```

In all the above cases, you would need a set of parentheses to bind a `*` symbol on the left between these `()` and `[]` right-side symbols in order for the declaration to be legal.

Here are some more examples:

Legal

```
int i;               an int
int *p;              an int pointer (ptr to an int)
int a[];             an array of ints
int f();             a function returning an int
int **pp;            a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];         a pointer to an array of ints
int (*pf)();         a pointer to a function returning an int
int *ap[];           an array of int pointers (array of ptrs to ints)
int aa[][];          an array of arrays of ints
int *fp();           a function returning an int pointer
int ***ppp;          a pointer to a pointer to an int pointer
int (**ppa)[];       a pointer to a pointer to an array of ints
int (**ppf)();       a pointer to a pointer to a function returning an int
int *(*pap)[];       a pointer to an array of int pointers
int (*paa)[][];      a pointer to an array of arrays of ints
int *(*pfp)();       a pointer to a function returning an int pointer
int **app[];         an array of pointers to int pointers
int (*apa[])[];      an array of pointers to arrays of ints
int (*apf[])();      an array of pointers to functions returning an int
int *aap[][];        an array of arrays of int pointers
int aaa[][][];       an array of arrays of arrays of int
int **fpp();         a function returning a pointer to an int pointer
int (*fpa())[];      a function returning a pointer to an array of ints
int (*fpf())();      a function returning a pointer to a function returning an int
```

Illegal

```
int af[]();          an array of functions returning an int
int fa()[];          a function returning an array of ints
int ff()();          a function returning a function returning an int
int (*pfa)()[];      a pointer to a function returning an array of ints
int aaf[][]();       an array of arrays of functions returning an int
int (*paf)[]();      a pointer to a an array of functions returning an int
int (*pff)()();      a pointer to a function returning a function returning an int
int *afp[]();        an array of functions returning int pointers
int afa[]()[];       an array of functions returning an array of ints
int aff[]()();       an array of functions returning functions returning an int
int *fap()[];        a function returning an array of int pointers
int faa()[][];       a function returning an array of arrays of ints
int faf()[]();       a function returning an array of functions returning an int
int *ffp()();        a function returning a function returning an int pointer
```

Source: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

Chapter 45: Structure Padding and Packing

By default, C compilers lay out structures so that each member can be accessed fast, without incurring penalties for 'unaligned access, a problem with RISC machines such as the DEC Alpha, and some ARM CPUs.

Depending on the CPU architecture and the compiler, a structure may occupy more space in memory than the sum of the sizes of its component members. The compiler can add padding between members or at the end of the structure, but not at the beginning.

Packing overrides the default padding.

Section 45.1: Packing structures

By default structures are padded in C. If you want to avoid this behaviour, you have to explicitly request it. Under GCC it's `__attribute__((__packed__))`. Consider this example on a 64-bit machine:

```
struct foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

The structure will be automatically padded to have`8-byte` alignment and will look like this:

```
struct foo {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte  */

    char pad[7]; /* 7 bytes added by compiler */

    long x;      /* 8 bytes */
};
```

So `sizeof(struct foo)` will give us 24 instead of 17. This happened because of a 64 bit compiler read/write from/to Memory in 8 bytes of word in each step and obvious when try to write `char c;` a one byte in memory a complete 8 bytes (i.e. word) fetched and consumes only first byte of it and its seven successive of bytes remains empty and not accessible for any read and write operation for structure padding.

Structure packing

But if you add the attribute `packed`, the compiler will not add padding:

```
struct __attribute__((__packed__)) foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

Now `sizeof(struct foo)` will return 17.

Generally packed structures are used:

- To save space.

- To format a data structure to transmit over network without depending on each architecture alignment of each node of the network.

It must be taken in consideration that some processors such as the ARM Cortex-M0 do not allow unaligned memory access; in such cases, structure packing can lead to *undefined behaviour* and can crash the CPU.

Section 45.2: Structure padding

Suppose this `struct` is defined and compiled with a 32 bit compiler:

```
struct test_32 {
    int a;      // 4 byte
    short b;    // 2 byte
    int c;      // 4 byte
} str_32;
```

We might expect this `struct` to occupy only 10 bytes of memory, but by printing `sizeof(str_32)` we see it uses 12 bytes.

This happened because the compiler aligns variables for fast access. A common pattern is that when the base type occupies N bytes (where N is a power of 2 such as 1, 2, 4, 8, 16 — and seldom any bigger), the variable should be aligned on an N-byte boundary (a multiple of N bytes).

For the structure shown with `sizeof(int) == 4` and `sizeof(short) == 2`, a common layout is:

- `int a;` stored at offset 0; size 4.
- `short b;` stored at offset 4; size 2.
- unnamed padding at offset 6; size 2.
- `int c;` stored at offset 8; size 4.

Thus `struct test_32` occupies 12 bytes of memory. In this example, there is no trailing padding.

The compiler will ensure that any `struct test_32` variables are stored starting on a 4-byte boundary, so that the members within the structure will be properly aligned for fast access. Memory allocation functions such as `malloc()`, `calloc()` and `realloc()` are required to ensure that the pointer returned is sufficiently well aligned for use with any data type, so dynamically allocated structures will be properly aligned too.

You can end up with odd situations such as on a 64-bit Intel x86_64 processor (e.g. Intel Core i7 — a Mac running macOS Sierra or Mac OS X), where when compiling in 32-bit mode, the compilers place `double` aligned on a 4-byte boundary; but, on the same hardware, when compiling in 64-bit mode, the compilers place `double` aligned on an 8-byte boundary.

Chapter 46: Memory management

name	description
size (malloc, realloc and aligned_alloc)	total size of the memory in bytes. For aligned_alloc the size must be a integral multiple of alignment.
size (calloc)	size of each element
nelements	number of elements
ptr	pointer to allocated memory previously returned by malloc, calloc, realloc or aligned_alloc
alignment	alignment of allocated memory

For managing dynamically allocated memory, the standard C library provides the functions malloc(), calloc(), realloc() and free(). In C99 and later, there is also aligned_alloc(). Some systems also provide alloca().

Section 46.1: Allocating Memory

Standard Allocation

The C dynamic memory allocation functions are defined in the **<stdlib.h>** header. If one wishes to allocate memory space for an object dynamically, the following code can be used:

```
int *p = malloc(10 * sizeof *p);
if (p == NULL)
{
    perror("malloc() failed");
    return -1;
}
```

This computes the number of bytes that ten ints occupy in memory, then requests that many bytes from malloc and assigns the result (i.e., the starting address of the memory chunk that was just created using malloc) to a pointer named p.

It is good practice to use sizeof to compute the amount of memory to request since the result of sizeof is implementation defined (except for *character types*, which are char, signed char and unsigned char, for which sizeof is defined to always give 1).

Because malloc might not be able to service the request, it might return a null pointer. It is important to check for this to prevent later attempts to dereference the null pointer.

Memory dynamically allocated using malloc() may be resized using realloc() or, when no longer needed, released using free().

Alternatively, declaring int array[10]; would allocate the same amount of memory. However, if it is declared inside a function without the keyword static, it will only be usable within the function it is declared in and the functions it calls (because the array will be allocated on the stack and the space will be released for reuse when the function returns). Alternatively, if it is defined with static inside a function, or if it is defined outside any function, then its lifetime is the lifetime of the program. Pointers can also be returned from a function, however a function in C can not return an array.

Zeroed Memory

The memory returned by malloc may not be initialized to a reasonable value, and care should be taken to zero the memory with memset or to immediately copy a suitable value into it. Alternatively, calloc returns a block of the

desired size where all bits are initialized to `0`. This need not be the same as the representation of floating-point zero or a null pointer constant.

```
int *p = calloc(10, sizeof *p);
if (p == NULL)
{
    perror("calloc() failed");
    return -1;
}
```

A note on `calloc`: Most (commonly used) implementations will optimise `calloc()` for performance, so it will be faster than calling `malloc()`, then `memset()`, even though the net effect is identical.

Aligned Memory

Version ≥ C11

C11 introduced a new function `aligned_alloc()` which allocates space with the given alignment. It can be used if the memory to be allocated is needed to be aligned at certain boundaries which can't be satisfied by `malloc()` or `calloc()`. `malloc()` and `calloc()` functions allocate memory that's suitably aligned for *any* object type (i.e. the alignment is `alignof(max_align_t)`). But with `aligned_alloc()` greater alignments can be requested.

```
/* Allocates 1024 bytes with 256 bytes alignment. */
char *ptr = aligned_alloc(256, 1024);
if (ptr) {
    perror("aligned_alloc()");
    return -1;
}
free(ptr);
```

The C11 standard imposes two restrictions: 1) the *size* (second argument) requested must be an integral multiple of the *alignment* (first argument) and 2) the value of *alignment* should be a valid alignment supported by the implementation. Failure to meet either of them results in undefined behavior.

Section 46.2: Freeing Memory

It is possible to release dynamically allocated memory by calling free().

```
int *p = malloc(10 * sizeof *p); /* allocation of memory */
if (p == NULL)
{
    perror("malloc failed");
    return -1;
}

free(p); /* release of memory */
/* note that after free(p), even using the *value* of the pointer p
   has undefined behavior, until a new value is stored into it. */

/* reusing/re-purposing the pointer itself */
int i = 42;
p = &i; /* This is valid, has defined behaviour */
```

The memory pointed to by `p` is reclaimed (either by the libc implementation or by the underlying OS) after the call to `free()`, so accessing that freed memory block via `p` will lead to undefined behavior. Pointers that reference memory elements that have been freed are commonly called dangling pointers, and present a security risk. Furthermore, the C standard states that even accessing the value of a dangling pointer has undefined behavior.

Note that the pointer p itself can be re-purposed as shown above.

Please note that you can only call free() on pointers that have directly been returned from the malloc(), calloc(), realloc() and aligned_alloc() functions, or where documentation tells you the memory has been allocated that way (functions like strdup () are notable examples). Freeing a pointer that is,

- obtained by using the & operator on a variable, or
- in the middle of an allocated block,

is forbidden. Such an error will usually not be diagnosed by your compiler but will lead the program execution in an undefined state.

There are two common strategies to prevent such instances of undefined behavior.

The first and preferable is simple - have p itself cease to exist when it is no longer needed, for example:

```
if (something_is_needed())
{

    int *p = malloc(10 * sizeof *p);
    if (p == NULL)
    {
        perror("malloc failed");
        return -1;
    }

    /* do whatever is needed with p */

    free(p);
}
```

By calling free() directly before the end of the containing block (i.e. the }), p itself ceases to exist. The compiler will give a compilation error on any attempt to use p after that.

A second approach is to also invalidate the pointer itself after releasing the memory to which it points:

```
free(p);
p = NULL;     // you may also use 0 instead of NULL
```

Arguments for this approach:

- On many platforms, an attempt to dereference a null pointer will cause instant crash: Segmentation fault. Here, we get at least a stack trace pointing to the variable that was used after being freed.

  Without setting pointer to NULL we have dangling pointer. The program will very likely still crash, but later, because the memory to which the pointer points will silently be corrupted. Such bugs are difficult to trace because they can result in a call stack that completely unrelated to the initial problem.

  This approach hence follows the fail-fast concept.

- It is safe to free a null pointer. The C Standard specifies that free(NULL) has no effect:

  > The free function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not

> match a pointer earlier returned by the `calloc`, `malloc`, or `realloc` function, or if the space has been deallocated by a call to `free` or `realloc`, the behavior is undefined.

- Sometimes the first approach cannot be used (e.g. memory is allocated in one function, and deallocated much later in a completely different function)

Section 46.3: Reallocating Memory

You may need to expand or shrink your pointer storage space after you have allocated memory to it. The `void *realloc(void *ptr, size_t size)` function deallocates the old object pointed to by `ptr` and returns a pointer to an object that has the size specified by `size`. `ptr` is the pointer to a memory block previously allocated with `malloc`, `calloc` or `realloc` (or a null pointer) to be reallocated. The maximal possible contents of the original memory is preserved. If the new size is larger, any additional memory beyond the old size are uninitialized. If the new size is shorter, the contents of the shrunken part is lost. If `ptr` is NULL, a new block is allocated and a pointer to it is returned by the function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = malloc(10 * sizeof *p);
    if (NULL == p)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    p[0] = 42;
    p[9] = 15;

    /* Reallocate array to a larger size, storing the result into a
     * temporary pointer in case realloc() fails. */
    {
        int *temporary = realloc(p, 1000000 * sizeof *temporary);

        /* realloc() failed, the original allocation was not free'd yet. */
        if (NULL == temporary)
        {
            perror("realloc() failed");
            free(p); /* Clean up. */
            return EXIT_FAILURE;
        }

        p = temporary;
    }

    /* From here on, array can be used with the new size it was
     * realloc'ed to, until it is free'd. */

    /* The values of p[0] to p[9] are preserved, so this will print:
       42 15
    */
    printf("%d %d\n", p[0], p[9]);

    free(p);
```

```
    return EXIT_SUCCESS;
}
```

The reallocated object may or may not have the same address as `*p`. Therefore it is important to capture the return value from `realloc` which contains the new address if the call is successful.

Make sure you assign the return value of `realloc` to a `temporary` instead of the original `p`. `realloc` will return null in case of any failure, which would overwrite the pointer. This would lose your data and create a memory leak.

Section 46.4: realloc(ptr, 0) is not equivalent to free(ptr)

`realloc` is *conceptually equivalent* to `malloc + memcpy + free` on the other pointer.

If the size of the space requested is zero, the behavior of `realloc` is implementation-defined. This is similar for all memory allocation functions that receive a `size` parameter of value `0`. Such functions may in fact return a non-null pointer, but that must never be dereferenced.

Thus, `realloc(ptr,0)` is not equivalent to `free(ptr)`. It may

- be a "lazy" implementation and just return `ptr`
- `free(ptr)`, allocate a dummy element and return that
- `free(ptr)` and return `0`
- just return `0` for failure and do nothing else.

So in particular the latter two cases are indistinguishable by application code.

This means `realloc(ptr,0)` may not really free/deallocate the memory, and thus it should never be used as a replacement for `free`.

Section 46.5: Multidimensional arrays of variable size

`Version ≥ C99`

Since C99, C has variable length arrays, VLA, that model arrays with bounds that are only known at initialization time. While you have to be careful not to allocate too large VLA (they might smash your stack), using *pointers to VLA* and using them in `sizeof` expressions is fine.

```
double sumAll(size_t n, size_t m, double A[n][m]) {
    double ret = 0.0;
    for (size_t i = 0; i < n; ++i)
       for (size_t j = 0; j < m; ++j)
          ret += A[i][j]
    return ret;
}

int main(int argc, char *argv[argc+1]) {
   size_t n = argc*10;
   size_t m = argc*8;
   double (*matrix)[m] = malloc(sizeof(double[n][m]));
   // initialize matrix somehow
   double res = sumAll(n, m, matrix);
   printf("result is %g\n", res);
   free(matrix);
}
```

Here `matrix` is a pointer to elements of type `double[m]`, and the `sizeof` expression with `double[n][m]` ensures that it contains space for `n` such elements.

All this space is allocated contiguously and can thus be deallocated by a single call to `free`.

The presence of VLA in the language also affects the possible declarations of arrays and pointers in function headers. Now, a general integer expression is permitted inside the `[]` of array parameters. For both functions the expressions in `[]` use parameters that have declared before in the parameter list. For `sumAll` these are the lengths that the user code expects for the matrix. As for all array function parameters in C the innermost dimension is rewritten to a pointer type, so this is equivalent to the declaration

```
double sumAll(size_t n, size_t m, double (*A)[m]);
```

That is, `n` is not really part of the function interface, but the information can be useful for documentation and it could also be used by bounds checking compilers to warn about out-of-bounds access.

Likwise, for `main`, the expression `argc+1` is the minimal length that the C standard prescribes for the `argv` argument.

Note that officially VLA support is optional in C11, but we know of no compiler that implements C11 and that doesn't have them. You could test with the macro `__STDC_NO_VLA__` if you must.

Section 46.6: alloca: allocate memory on stack

Caveat: `alloca` is only mentioned here for the sake of completeness. It is entirely non-portable (not covered by any of the common standards) and has a number of potentially dangerous features that make it un-safe for the unaware. Modern C code should replace it with *Variable Length Arrays* (VLA).

Manual page

```
#include <alloca.h>
// glibc version of stdlib.h include alloca.h by default

void foo(int size) {
    char *data = alloca(size);
    /*
      function body;
    */
    // data is automatically freed
}
```

Allocate memory on the stack frame of the caller, the space referenced by the returned pointer is automatically free'd when the caller function finishes.

While this function is convenient for automatic memory management, be aware that requesting large allocation could cause a stack overflow, and that you cannot use `free` with memory allocated with `alloca` (which could cause more issue with stack overflow).

For these reason it is not recommended to use `alloca` inside a loop nor a recursive function.

And because the memory is `free`'d upon function return you cannot return the pointer as a function result (the behavior would be undefined).

Summary

- call identical to `malloc`
- automatically free'd upon function return
- incompatible with `free`,`realloc` functions (undefined behavior)
- pointer cannot be returned as a function result (undefined behavior)

- allocation size limited by stack space, which (on most machines) is a lot smaller than the heap space available for use by `malloc()`
- avoid using `alloca()` and VLAs (variable length arrays) in a single function
- `alloca()` is not as portable as `malloc()` et al

Recommendation

- Do not use `alloca()` in new code

Version ≥ C99

Modern alternative.

```
void foo(int size) {
    char data[size];
    /*
      function body;
    */
    // data is automatically freed
}
```

This works where `alloca()` does, and works in places where `alloca()` doesn't (inside loops, for example). It does assume either a C99 implementation or a C11 implementation that does not define `__STDC_NO_VLA__`.

Section 46.7: User-defined memory management

`malloc()` often calls underlying operating system functions to obtain pages of memory. But there is nothing special about the function and it can be implemented in straight C by declaring a large static array and allocating from it (there is a slight difficulty in ensuring correct alignment, in practice aligning to 8 bytes is almost always adequate).

To implement a simple scheme, a control block is stored in the region of memory immediately before the pointer to be returned from the call. This means that `free()` may be implemented by subtracting from the returned pointer and reading off the control information, which is typically the block size plus some information that allows it to be put back in the free list - a linked list of unallocated blocks.

When the user requests an allocation, the free list is searched until a block of identical or larger size to the amount requested is found, then if necessary it is split. This can lead to memory fragmentation if the user is continually making many allocations and frees of unpredictable size and and at unpredictable intervals (not all real programs behave like that, the simple scheme is often adequate for small programs).

```
/* typical control block */
struct block
{
   size_t size;         /* size of block */
   struct block *next;  /* next block in free list */
   struct block *prev;  /* back pointer to previous block in memory */
   void *padding;       /* need 16 bytes to make multiple of 8 */
}

static struct block arena[10000]; /* allocate from here */
static struct block *firstfree;
```

Many programs require large numbers of allocations of small objects of the same size. This is very easy to implement. Simply use a block with a next pointer. So if a block of 32 bytes is required:

```
union block
```

```
{
   union block * next;
   unsigned char payload[32];
}

static union block arena[100];
static union block * head;
void init(void)
{
    int i;
    for (i = 0; i < 100 - 1; i++)
        arena[i].next = &arena[i + 1];
    arena[i].next = 0; /* last one, null */
    head = &block[0];
}

void *block_alloc()
{
    void *answer = head;
    if (answer)
        head = head->next;
    return answer;
}

void block_free(void *ptr)
{
    union block *block = ptr;
    block->next = head;
    head - block;
}
```

This scheme is extremely fast and efficient, and can be made generic with a certain loss of clarity.

Chapter 47: Implementation-defined behaviour

Section 47.1: Right shift of a negative integer

```
int signed_integer = -1;

// The right shift operation exhibits implementation-defined behavior:
int result = signed_integer >> 1;
```

Section 47.2: Assigning an out-of-range value to an integer

```
// Supposing SCHAR_MAX, the maximum value that can be represented by a signed char, is
// 127, the behavior of this assignment is implementation-defined:
signed char integer;
integer = 128;
```

Section 47.3: Allocating zero bytes

```
// The allocation functions have implementation-defined behavior when the requested size
// of the allocation is zero.
void *p = malloc(0);
```

Section 47.4: Representation of signed integers

Each signed integer type may be represented in any one of three formats; it is implementation-defined which one is used. The implementation in use for any given signed integer type at least as wide as `int` can be determined at runtime from the two lowest-order bits of the representation of value `-1` in that type, like so:

```
enum { sign_magnitude = 1, ones_compl = 2, twos_compl = 3, };
#define SIGN_REP(T) ((T)-1 & (T)3)

switch (SIGN_REP(long)) {
   case sign_magnitude: { /* do something */ break; }
   case ones_compl:     { /* do otherwise */ break; }
   case twos_compl:     { /* do yet else  */ break; }
   case 0:  { _Static_assert(SIGN_REP(long), "bogus sign representation"); }
}
```

The same pattern applies to the representation of narrower types, but they cannot be tested by this technique because the operands of `&` are subject to "the usual arithmetic conversions" before the result is computed.

Chapter 48: Atomics

Section 48.1: atomics and operators

Atomic variables can be accessed concurrently between different threads without creating race conditions.

```
/* a global static variable that is visible by all threads */
static unsigned _Atomic active = ATOMIC_VAR_INIT(0);


int myThread(void* a) {
  ++active;         // increment active race free
  // do something
  --active;         // decrement active race free
  return 0;
}
```

All lvalue operations (operations that modify the object) that are allowed for the base type are allowed and will not lead to race conditions between different threads that access them.

- Operations on atomic objects are generally orders of magnitude slower than normal arithmetic operations. This also includes simple load or store operations. So you should only use them for critical tasks.
- Usual arithmetic operations and assignment such as `a = a+1;` are in fact three operations on `a`: first a load, then addition and finally a store. This is *not* race free. Only the operation `a += 1;` and `a++;` are.

Chapter 49: Jump Statements

Section 49.1: Using return

Returning a value

One commonly used case: returning from `main()`

```
#include <stdlib.h> /* for EXIT_xxx macros */

int main(int argc, char ** argv)
{
  if (2 < argc)
  {
    return EXIT_FAILURE; /* The code expects one argument:
                            leave immediately skipping the rest of the function's code */
  }

  /* Do stuff. */

  return EXIT_SUCCESS;
}
```

Additional notes:

1. For a function having a return type as `void` (not including `void *` or related types), the `return` statement should not have any associated expression; i.e, the only allowed return statement would be `return;`.
2. For a function having a non-`void` return type, the `return` statement shall not appear without an expression.
3. For `main()` (and only for `main()`), an *explicit* `return` statement is not required (in C99 or later). If the execution reaches the terminating `}`, an implicit value of `0` is returned. Some people think omitting this `return` is bad practice; others actively suggest leaving it out.

Returning nothing

Returning from a `void` function

```
void log(const char * message_to_log)
{
  if (NULL == message_to_log)
  {
    return; /* Nothing to log, go home NOW, skip the logging. */
  }

  fprintf(stderr, "%s:%d %s\n", __FILE__, _LINE__, message_to_log);

  return; /* Optional, as this function does not return a value. */
}
```

Section 49.2: Using goto to jump out of nested loops

Jumping out of nested loops would usually require use of a boolean variable with a check for this variable in the loops. Supposing we are iterating over `i` and `j`, it could look like this

```
size_t i,j;
for (i = 0; i < myValue && !breakout_condition; ++i) {
```

```
    for (j = 0; j < mySecondValue && !breakout_condition; ++j) {
        ... /* Do something, maybe modifying breakout_condition */
            /* When breakout_condition == true the loops end */
     }
}
```

But the C language offers the `goto` clause, which can be useful in this case. By using it with a label declared after the loops, we can easily break out of the loops.

```
size_t i,j;
for (i = 0; i < myValue; ++i) {
    for (j = 0; j < mySecondValue; ++j) {
        ...
        if(breakout_condition)
          goto final;
    }
}
final:
```

However, often when this need comes up a `return` could be better used instead. This construct is also considered "unstructured" in structural programming theory.

Another situation where `goto` might be useful is for jumping to an error-handler:

```
ptr = malloc(N *  x);
if(!ptr)
  goto out_of_memory;

/* normal processing */
free(ptr);
return SUCCESS;

out_of_memory:
 free(ptr); /* harmless, and necessary if we have further errors */
 return FAILURE;
```

Use of `goto` keeps error flow separate from normal program control flow. It is however also considered "unstructured" in the technical sense.

Section 49.3: Using break and continue

Immediately `continue` reading on invalid input or **`break`** on user request or end-of-file:

```
#include <stdlib.h> /* for EXIT_xxx macros */
#include <stdio.h>  /* for printf() and getchar() */
#include <ctype.h> /* for isdigit() */

void flush_input_stream(FILE * fp);


int main(void)
{
  int sum = 0;
  printf("Enter digits to be summed up or 0 to exit:\n");

  do
  {
    int c = getchar();
    if (EOF == c)
```

```
    {
      printf("Read 'end-of-file', exiting!\n");

      break;
    }

    if ('\n' != c)
    {
      flush_input_stream(stdin);
    }

    if (!isdigit(c))
    {
      printf("%c is not a digit! Start over!\n", c);

      continue;
    }

    if ('0' == c)
    {
      printf("Exit requested.\n");

      break;
    }

    sum += c - '0';

    printf("The current sum is %d.\n", sum);
  } while (1);

  return EXIT_SUCCESS;
}

void flush_input_stream(FILE * fp)
{
  size_t i = 0;
  int c;
  while ((c = fgetc(fp)) != '\n' && c != EOF) /* Pull all until and including the next new-line. */
  {
    ++i;
  }

  if (0 != i)
  {
    fprintf(stderr, "Flushed %zu characters from input.\n", i);
  }
}
```

Chapter 50: Create and include header files

In modern C, header files are crucial tools that must be designed and used correctly. They allow the compiler to cross-check independently compiled parts of a program.

Headers declare types, functions, macros etc that are needed by the consumers of a set of facilities. All the code that uses any of those facilities includes the header. All the code that defines those facilities includes the header. This allows the compiler to check that the uses and definitions match.

Section 50.1: Introduction

There are a number of guidelines to follow when creating and using header files in a C project:

- Idemopotence

  If a header file is included multiple times in a translation unit (TU), it should not break builds.

- Self-containment

  If you need the facilities declared in a header file, you should not have to include any other headers explicitly.

- Minimality

  You should not be able to remove any information from a header without causing builds to fail.

- Include What You Use (IWYU)

  Of more concern to C++ than C, but nevertheless important in C too. If the code in a TU (call it `code.c`) directly uses the features declared by a header (call it `"headerA.h"`), then `code.c` should `#include "headerA.h"` directly, even if the TU includes another header (call it `"headerB.h"`) that happens, at the moment, to include `"headerA.h"`.

Occasionally, there might be good enough reasons to break one or more of these guidelines, but you should both be aware that you are breaking the rule and be aware of the consequences of doing so before you break it.

Section 50.2: Self-containment

Modern headers should be self-contained, which means that a program that needs to use the facilities defined by `header.h` can include that header (`#include "header.h"`) and not worry about whether other headers need to be included first.

Recommendation: Header files should be self-contained.

Historical rules

Historically, this has been a mildly contentious subject.

Once upon another millennium, the AT&T Indian Hill C Style and Coding Standards stated:

> Header files should not be nested. The prologue for a header file should, therefore, describe what other headers need to be `#include`d for the header to be functional. In extreme cases, where a large number of header files are to be included in several different source files, it is acceptable to put all common `#include`s in one include file.

This is the antithesis of self-containment.

Modern rules

However, since then, opinion has tended in the opposite direction. If a source file needs to use the facilities declared by a header `header.h`, the programmer should be able to write:

```
#include "header.h"
```

and (subject only to having the correct search paths set on the command line), any necessary pre-requisite headers will be included by `header.h` without needing any further headers added to the source file.

This provides better modularity for the source code. It also protects the source from the "guess why this header was added" conundrum that arises after the code has been modified and hacked for a decade or two.

The NASA Goddard Space Flight Center (GSFC) coding standards for C is one of the more modern standards — but is now a little hard to track down. It states that headers should be self-contained. It also provides a simple way to ensure that headers are self-contained: the implementation file for the header should include the header as the first header. If it is not self-contained, that code will not compile.

The rationale given by GSFC includes:

> §2.1.1 Header include rationale

> This standard requires a unit's header to contain `#include` statements for all other headers required by the unit header. Placing `#include` for the unit header first in the unit body allows the compiler to verify that the header contains all required `#include` statements.

> An alternate design, not permitted by this standard, allows no `#include` statements in headers; all `#includes` are done in the body files. Unit header files then must contain #ifdef statements that check that the required headers are included in the proper order.

> One advantage of the alternate design is that the `#include` list in the body file is exactly the dependency list needed in a makefile, and this list is checked by the compiler. With the standard design, a tool must be used to generate the dependency list. However, all of the branch recommended development environments provide such a tool.

> A major disadvantage of the alternate design is that if a unit's required header list changes, each file that uses that unit must be edited to update the `#include` statement list. Also, the required header list for a

> compiler library unit may be different on different targets.

> Another disadvantage of the alternate design is that compiler library header files, and other third party files, must be modified to add the required `#ifdef` statements.

Thus, self-containment means that:

- If a header `header.h` needs a new nested header `extra.h`, you do not have to check every source file that uses `header.h` to see whether you need to add `extra.h`.
- If a header `header.h` no longer needs to include a specific header `notneeded.h`, you do not have to check every source file that uses `header.h` to see whether you can safely remove `notneeded.h` (but see Include what you use.
- You do not have to establish the correct sequence for including the pre-requisite headers (which requires a topological sort to do the job properly).

Checking self-containment

See Linking against a static library for a script `chkhdr` that can be used to test idempotence and self-containment of a header file.

Section 50.3: Minimality

Headers are a crucial consistency checking mechanism, but they should be as small as possible. In particular, that means that a header should not include other headers just because the implementation file will need the other headers. A header should contain only those headers necessary for a consumer of the services described.

For example, a project header should not include **`<stdio.h>`** unless one of the function interfaces uses the type `FILE *` (or one of the other types defined solely in **`<stdio.h>`**). If an interface uses `size_t`, the smallest header that suffices is **`<stddef.h>`**. Obviously, if another header that defines `size_t` is included, there is no need to include **`<stddef.h>`** too.

If the headers are minimal, then it keeps the compilation time to a minimum too.

It is possible to devise headers whose sole purpose is to include a lot of other headers. These seldom turn out to be a good idea in the long run because few source files will actually need all the facilities described by all the headers. For example, a **`<standard-c.h>`** could be devised that includes all the standard C headers — with care since some headers are not always present. However, very few programs actually use the facilities of **`<locale.h>`** or **`<tgmath.h>`**.

- See also How to link multiple implementation files in C?

Section 50.4: Notation and Miscellany

The C standard says that there is very little difference between the `#include <header.h>` and `#include "header.h"` notations.

> [`#include <header.h>`] searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the `<` and `>` delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

> [`#include "header.h"`] causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"..."` delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read [`#include <header.h>`] …

So, the double quoted form may look in more places than the angle-bracketed form. The standard specifies by example that the standard headers should be included in angle-brackets, even though the compilation works if you use double quotes instead. Similarly, standards such as POSIX use the angle-bracketed format — and you should too. Reserve double-quoted headers for headers defined by the project. For externally-defined headers (including headers from other projects your project relies on), the angle-bracket notation is most appropriate.

Note that there should be a space between `#include` and the header, even though the compilers will accept no space there. Spaces are cheap.

A number of projects use a notation such as:

```
#include <openssl/ssl.h>
#include <sys/stat.h>
#include <linux/kernel.h>
```

You should consider whether to use that namespace control in your project (it is quite probably a good idea). You should steer clear of the names used by existing projects (in particular, both `sys` and `linux` would be bad choices).

If you use this, your code should be careful and consistent in the use of the notation.

Do not use `#include "../include/header.h"` notation.

Header files should seldom if ever define variables. Although you will keep global variables to a minimum, if you need a global variable, you will declare it in a header, and define it in one suitable source file, and that source file will include the header to cross-check the declaration and definition, and all source files that use the variable will use the header to declare it.

Corollary: you will not declare global variables in a source file — a source file will only contain definitions.

Header files should seldom declare `static` functions, with the notable exception of `static inline` functions which will be defined in headers if the function is needed in more than one source file.

- Source files define global variables, and global functions.
- Source files do not declare the existence of global variables or functions; they include the header that declares the variable or function.
- Header files declare global variable and functions (and types and other supporting material).
- Header files do not define variables or any functions except (`static`) `inline` functions.

Cross-references

- Where to document functions in C?
- List of standard header files in C and C++
- Is `inline` without `static` or `extern` ever useful in C99?
- How do I use `extern` to share variables between source files?
- What are the benefits of a relative path such as `"../include/header.h"` for a header?
- Header inclusion optimization
- Should I include every header?

Section 50.5: Idempotence

If a particular header file is included more than once in a translation unit (TU), there should not be any compilation problems. This is termed 'idempotence'; your headers should be idempotent. Think how difficult life would be if you had to ensure that `#include <stdio.h>` was only included once.

There are two ways to achieve idempotence: header guards and the `#pragma once` directive.

Header guards

Header guards are simple and reliable and conform to the C standard. The first non-comment lines in a header file should be of the form:

```
#ifndef UNIQUE_ID_FOR_HEADER
#define UNIQUE_ID_FOR_HEADER
```

The last non-comment line should be `#endif`, optionally with a comment after it:

```
#endif /* UNIQUE_ID_FOR_HEADER */
```

All the operational code, including other `#include` directives, should be between these lines.

Each name must be unique. Often, a name scheme such as `HEADER_H_INCLUDED` is used. Some older code uses a symbol defined as the header guard (e.g. `#ifndef BUFSIZ` in **`<stdio.h>`**), but it is not as reliable as a unique name.

One option would be to use a generated MD5 (or other) hash for the header guard name. You should avoid emulating the schemes used by system headers which frequently use names reserved to the implementation — names starting with an underscore followed by either another underscore or an upper-case letter.

The `#pragma once` Directive

Alternatively, some compilers support the `#pragma once` directive which has the same effect as the three lines shown for header guards.

```
#pragma once
```

The compilers which support `#pragma once` include MS Visual Studio and GCC and Clang. However, if portability is a concern, it is better to use header guards, or use both. Modern compilers (those supporting C89 or later) are required to ignore, without comment, pragmas that they do not recognize ('Any such pragma that is not recognized by the implementation is ignored') but old versions of GCC were not so indulgent.

Section 50.6: Include What You Use (IWYU)

Google's [Include What You Use] project, or IWYU, ensures source files include all headers used in the code.

Suppose a source file `source.c` includes a header `arbitrary.h` which in turn coincidentally includes `freeloader.h`, but the source file also explicitly and independently uses the facilities from `freeloader.h`. All is well to start with. Then one day `arbitrary.h` is changed so its clients no longer need the facilities of `freeloader.h`. Suddenly, `source.c` stops compiling — because it didn't meet the IWYU criteria. Because the code in `source.c` explicitly used the facilities of `freeloader.h`, it should have included what it uses — there should have been an explicit `#include "freeloader.h"` in the source too. (Idempotency would have ensured there wasn't a problem.)

The IWYU philosophy maximizes the probability that code continues to compile even with reasonable changes made to interfaces. Clearly, if your code calls a function that is subsequently removed from the published interface,

no amount of preparation can prevent changes becoming necessary. This is why changes to APIs are avoided when possible, and why there are deprecation cycles over multiple releases, etc.

This is a particular problem in C++ because standard headers are allowed to include each other. Source file `file.cpp` could include one header `header1.h` that on one platform includes another header `header2.h`. `file.cpp` might turn out to use the facilities of `header2.h` as well. This wouldn't be a problem initially - the code would compile because `header1.h` includes `header2.h`. On another platform, or an upgrade of the current platform, `header1.h` could be revised so it no longer includes `header2.h`, and then`file.cpp` would stop compiling as a result.

IWYU would spot the problem and recommend that `header2.h` be included directly in `file.cpp`. This would ensure it continues to compile. Analogous considerations apply to C code too.

Chapter 51: <ctype.h> — character classification & conversion

Section 51.1: Introduction

The header `ctype.h` is a part of the standard C library. It provides functions for classifying and converting characters.

All of these functions take one parameter, an `int` that must be either *EOF* or representable as an unsigned char.

The names of the classifying functions are prefixed with 'is'. Each returns an integer non-zero value (TRUE) if the character passed to it satisfies the related condition. If the condition is not satisfied then the function returns a zero value (FALSE).

These classifying functions operate as shown, assuming the default C locale:

```
int a;
int c = 'A';
a = isalpha(c); /* Checks if c is alphabetic (A-Z, a-z), returns non-zero here. */
a = isalnum(c); /* Checks if c  is alphanumeric (A-Z, a-z, 0-9), returns non-zero here. */
a = iscntrl(c); /* Checks is c is a control character (0x00-0x1F, 0x7F), returns zero here. */
a = isdigit(c); /* Checks if c is a digit (0-9), returns zero here. */
a = isgraph(c); /* Checks if c has a graphical representation (any printing character except space),
returns non-zero here. */
a = islower(c); /* Checks if c is a lower-case letter (a-z), returns zero here. */
a = isprint(c); /* Checks if c is any printable character (including space), returns non-zero here.
*/
a = isupper(c); /* Checks if c is a upper-case letter (a-z), returns zero here. */
a = ispunct(c); /* Checks if c is a punctuation character, returns zero here. */
a = isspace(c); /* Checks if c is a white-space character, returns zero here. */
a = isupper(c); /* Checks if c is an upper-case letter (A-Z), returns non-zero here. */
a = isxdigit(c); /* Checks if c is a hexadecimal digit (A-F, a-f, 0-9), returns non-zero here. */
```

Version ≥ C99

```
a = isblank(c); /* Checks if c is a blank character (space or tab), returns non-zero here. */
```

There are two conversion functions. These are named using the prefix 'to'. These functions take the same argument as those above. However the return value is not a simple zero or non-zero but the passed argument changed in some manner.

These conversion functions operate as shown, assuming the default C locale:

```
int a;
int c = 'A';

/* Converts c to a lower-case letter (a-z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'a' here.
 */
a = tolower(c);

/* Converts c to an upper-case letter (A-Z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'A' here.
 */
a = toupper(c);
```

The below information is quoted from cplusplus.com mapping how the original 127-character ASCII set is considered by each of the classifying type functions (a • indicates that the function returns non-zero for that character)

ASCII values	characters	iscntrl	isblank	isspace	isupper	islower	isalpha	isdigit	isxdigit	isalnum	ispunct	isgraph	isprint
0x00 .. 0x08	NUL, (other control codes)	•											
0x09	tab ('\t')	•	•	•									
0x0A .. 0x0D	(white-space control codes: '\f','\v','\n','\r')	•		•									
0x0E .. 0x1F	(other control codes)	•											
0x20	space (' ')		•	•									•
0x21 .. 0x2F	!"#$%&'()*+,-./										•	•	•
0x30 .. 0x39	0123456789							•	•	•		•	•
0x3a .. 0x40	:;<=>?@										•	•	•
0x41 .. 0x46	ABCDEF				•		•		•	•		•	•
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ				•		•			•		•	•
0x5B .. 0x60	[]^_`										•	•	•
0x61 .. 0x66	abcdef					•	•		•	•		•	•
0x67 .. 0x7A	ghijklmnopqrstuvwxyz					•	•			•		•	•
0x7B .. 0x7E	{}~bar										•	•	•
0x7F	(DEL)	•											

Section 51.2: Classifying characters read from a stream

```
#include <ctype.h>
#include <stdio.h>

typedef struct {
  size_t space;
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(FILE *f) {
  chartypes types = { 0, 0, 0 };
  int ch;

  while ((ch = fgetc(f)) != EOF) {
    types.space += !!isspace(ch);
    types.alnum += !!isalnum(ch);
    types.punct += !!ispunct(ch);
  }

  return types;
}
```

The `classify` function reads characters from a stream and counts the number of spaces, alphanumeric and

punctuation characters. It avoids several pitfalls.

- When reading a character from a stream, the result is saved as an `int`, since otherwise there would be an ambiguity between reading `EOF` (the end-of-file marker) and a character that has the same bit pattern.
- The character classification functions (e.g. `isspace`) expect their argument to be *either representable as an `unsigned char`, or the value of the `EOF` macro*. Since this is exactly what the `fgetc` returns, there is no need for conversion here.
- The return value of the character classification functions only distinguishes between zero (meaning **`false`**) and nonzero (meaning **`true`**). For counting the number of occurrences, this value needs to be converted to a 1 or 0, which is done by the double negation, `!!`.

Section 51.3: Classifying characters from a string

```
#include <ctype.h>
#include <stddef.h>

typedef struct {
  size_t space;
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(const char *s) {
  chartypes types = { 0, 0, 0 };
  const char *p;
  for (p= s; p != '\0'; p++) {
    types.space += !!isspace((unsigned char)*p);
    types.alnum += !!isalnum((unsigned char)*p);
    types.punct += !!ispunct((unsigned char)*p);
  }

  return types;
}
```

The `classify` function examines all characters from a string and counts the number of spaces, alphanumeric and punctuation characters. It avoids several pitfalls.

- The character classification functions (e.g. `isspace`) expect their argument to be *either representable as an `unsigned char`, or the value of the `EOF` macro*.
- The expression `*p` is of type `char` and must therefore be converted to match the above wording.
- The `char` type is defined to be equivalent to either `signed char` or `unsigned char`.
- When `char` is equivalent to `unsigned char`, there is no problem, since every possible value of the `char` type is representable as `unsigned char`.
- When `char` is equivalent to `signed char`, it must be converted to `unsigned char` before being passed to the character classification functions. And although the value of the character may change because of this conversion, this is exactly what these functions expect.
- The return value of the character classification functions only distinguishes between zero (meaning **`false`**) and nonzero (meaning **`true`**). For counting the number of occurrences, this value needs to be converted to a 1 or 0, which is done by the double negation, `!!`.

Chapter 52: Side Effects

Section 52.1: Pre/Post Increment/Decrement operators

In C, there are two unary operators - '++' and '--' that are very common source of confusion. The operator **++** is called the *increment operator* and the operator `--` is called the *decrement operator*. Both of them can be used used in either *prefix* form or *postfix* form. The syntax for prefix form for **++** operator is `++operand` and the syntax for postfix form is `operand++`. When used in the prefix form, the operand is incremented first by `1` and the resultant value of the operand is used in the evaluation of the expression. Consider the following example:

```
int n, x = 5;
n = ++x; /* x is incremented  by 1(x=6), and result is assigned to n(6) */
         /* this is a short form for two statements: */
         /* x = x + 1; */
         /* n = x ; */
```

When used in the postfix form, the operand's current value is used in the expression and then the value of the operand is incremented by `1`. Consider the following example:

```
int n, x = 5;
n = x++; /* value of x(5) is assigned first to n(5), and then x is incremented by 1; x(6) */
         /* this is a short form for two statements: */
         /* n = x; */
         /* x = x + 1; */
```

The working of the decrement operator `--` can be understood similarly.

The following code demonstrates what each one does

```
    int main()
    {
        int a, b, x = 42;
        a = ++x; /* a and x are 43 */
        b = x++; /* b is 43, x is 44 */
        a = x--; /* a is is 44, x is 43 */
        b = --x; /* b and x are 42 */

        return 0;
    }
```

From the above it is clear that post operators return the current value of a variable and *then* modify it, but pre operators modify the variable and *then* return the modified value.

In all versions of C, the order of evaluation of pre and post operators are not defined, hence the following code can return unexpected outputs:

```
    int main()
    {
        int a, x = 42;
        a = x++ + x; /* wrong */
        a = x + x; /* right */
        ++x;

        int ar[10];
        x = 0;
        ar[x] = x++; /* wrong */
```

```
        ar[x++] = x; /* wrong */
        ar[x] = x; /* right */
        ++x;
        return 0;
    }
```

Note that it is also good practice to use pre over post operators when used alone in a statement. Look at the above code for this.

Note also, that when a function is called, all side effects on arguments must take place before the function runs.

```
    int foo(int x)
    {
        return x;
    }

    int main()
    {
        int a = 42;
        int b = foo(a++);  /* This returns 43, even if it seems like it should return 42 */
        return 0;
    }
```

Chapter 53: Multi-Character Character Sequence

Section 53.1: Trigraphs

The symbols `[ ] { } ^ \ | ~ #` are frequently used in C programs, but in the late 1980s, there were code sets in use (ISO 646 variants, for example, in Scandinavian countries) where the ASCII character positions for these were used for national language variant characters (e.g. `£` for `#` in the UK; `Æ Å æ å ø Ø` for `{ } { } | \` in Denmark; there was no `~` in EBCDIC). This meant that it was hard to write C code on machines that used these sets.

To solve this problem, the C standard suggested the use of combinations of three characters to produce a single character called a trigraph. A trigraph is a sequence of three characters, the first two of which are question marks.

The following is a simple example that uses trigraph sequences instead of `#`, `{` and `}`:

```
??=include <stdio.h>

int main()
??<
    printf("Hello World!\n");
??>
```

This will be changed by the C preprocessor by replacing the trigraphs with their single-character equivalents as if the code had been written:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
}
```

Trigraph	Equivalent
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

Note that trigraphs are problematic because, for example, `??/` is a backslash and can affect the meaning of continuation lines in comments, and have to be recognized inside strings and character literals (e.g. `'??/??/'` is a single character, a backslash).

Section 53.2: Digraphs

Version ≥ C99

In 1994 more readable alternatives to five of the trigraphs were supplied. These use only two characters and are known as digraphs. Unlike trigraphs, digraphs are tokens. If a digraph occurs in another token (e.g. string literals or

character constants) then it will not be treated as a digraph, but remain as it is.

The following shows the difference before and after processing the digraphs sequence.

```
#include <stdio.h>

int main()
<%
    printf("Hello %> World!\n"); /* Note that the string contains a digraph */
%>
```

Which will be treated the same as:

```
#include <stdio.h>

int main()
{
    printf("Hello %> World!\n"); /* Note the unchanged digraph within the string. */
}
```

Digraph	Equivalent
<:	[
:>	]
<%	{
%>	}
%:	#

Chapter 54: Constraints

Section 54.1: Duplicate variable names in the same scope

An example of a constraint as expressed in the C standard is having two variables of the same name declared in a scope1), for example:

```
void foo(int bar)
{
    int var;
    double var;
}
```

This code breaches the constraint and must produce a diagnostic message at compile time. This is very useful as compared to undefined behavior as the developer will be informed of the issue before the program is run, potentially doing anything.

Constraints thus tend to be errors which are easily detectable at compile time such as this, issues which result in undefined behavior but would be difficult or impossible to detect at compile time are thus not constraints.

1) exact wording:

```
Version = C99
```

If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified in 6.7.2.3.

Section 54.2: Unary arithmetic operators

The unary `+` and `-` operators are only usable on arithmetic types, therefore if for example one tries to use them on a struct the program will produce a diagnostic eg:

```
struct foo
{
    bool bar;
};

void baz(void)
{
   struct foo testStruct;
   -testStruct; /* This breaks the constraint so must produce a diagnostic */
}
```

Chapter 55: Inlining

Section 55.1: Inlining functions used in more than one source file

For small functions that get called often, the overhead associated with the function call can be a significant fraction of the total execution time of that function. One way of improving performance, then, is to eliminate the overhead.

In this example we use four functions (plus `main()`) in three source files. Two of those (`plusfive()` and `timestwo()`) each get called by the other two located in "source1.c" and "source2.c". The `main()` is included so we have a working example.

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int main(void) {
    int start = 3;
    int intermediate = complicated1(start);
    printf("First result is %d\n", intermediate);
    intermediate = complicated2(start);
    printf("Second result is %d\n", intermediate);
    return 0;
}
```

source1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated1(int input) {
    int tmp = timestwo(input);
    tmp = plusfive(tmp);
    return tmp;
}
```

source2.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated2(int input) {
    int tmp = plusfive(input);
    tmp = timestwo(tmp);
    return tmp;
}
```

headerfile.h:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H

int complicated1(int input);
int complicated2(int input);

inline int timestwo(int input) {
  return input * 2;
}
inline int plusfive(int input) {
```

```
  return input + 5;
}

#endif
```

Functions `timestwo` and `plusfive` get called by both `complicated1` and `complicated2`, which are in different "translation units", or source files. In order to use them in this way, we have to define them in the header.

Compile like this, assuming gcc:

```
cc -O2 -std=c99 -c -o main.o main.c
cc -O2 -std=c99 -c -o source1.o source1.c
cc -O2 -std=c99 -c -o source2.o source2.c
cc main.o source1.o source2.o -o main
```

We use the -O2 optimization option because some compilers don't inline without optimization turned on.

The effect of the **`inline`** keyword is that the function symbol in question is not emitted into the object file. Otherwise an error would occur in the last line, where we are linking the object files to form the final executable. If we would not have **`inline`**, the same symbol would be defined in both `.o` files, and a "multiply defined symbol" error would occur.

In situations where the symbol is actually needed, this has the disadvantage that the symbol is not produced at all. There are two possibilities to deal with that. The first is to add an extra **`extern`** declaration of the inlined functions in exactly one of the `.c` files. So add the following to `source1.c`:

```
extern int timestwo(int input);
extern int plusfive(int input);
```

The other possibility is to define the function with `static inline` instead of **`inline`**. This method has the drawback that eventually a copy of the function in question may be produced in **every** object file that is produced with this header.

Chapter 56: Unions

Section 56.1: Using unions to reinterpret values

Some C implementations permit code to write to one member of a union type then read from another in order to perform a sort of reinterpreting cast (parsing the new type as the bit representation of the old one).

It is important to note however, this is not permitted by the C standard current or past and will result in undefined behavior, none the less is is a very common extension offered by compilers (so check your compiler docs if you plan to do this).

One real life example of this technique is the "Fast Inverse Square Root" algorithm which relies on implementation details of IEEE 754 floating point numbers to perform an inverse square root more quickly than using floating point operations, this algorithm can be performed either through pointer casting (which is very dangerous and breaks the strict aliasing rule) or through a union (which is still undefined behavior but works in many compilers):

```
union floatToInt
{
    int32_t intMember;
    float floatMember; /* Float must be 32 bits IEEE 754 for this to work */
};

float inverseSquareRoot(float input)
{
    union floatToInt x;
    int32_t i;
    float f;
    x.floatMember = input;     /* Assign to the float member */
    i = x.intMember;           /* Read back from the integer member */
    i = 0x5f3759df - (i >> 1);
    x.intMember = i;           /* Assign to the integer member */
    f = x.floatMember;         /* Read back from the float member */
    f = f * (1.5f - input * 0.5f * f * f);
    return f * (1.5f - input * 0.5f * f * f);
}
```

This technique was widely used in computer graphics and games in the past due to its greater speed compared to using floating point operations, and is very much a compromise, losing some accuracy and being very non portable in exchange for speed.

Section 56.2: Writing to one union member and reading from another

The members of a union share the same space in memory. This means that writing to one member overwrites the data in all other members and that reading from one member results in the same data as reading from all other members. However, because union members can have differing types and sizes, the data that is read can be interpreted differently, see
http://stackoverflow.com/documentation/c/1119/structs-and-unions/9399/using-unions-to-reinterpret-values

The simple example below demonstrates a union with two members, both of the same type. It shows that writing to member `m_1` results in the written value being read from member `m_2` and writing to member `m_2` results in the written value being read from member `m_1`.

```
#include <stdio.h>
```

```
union my_union /* Define union */
{
    int m_1;
    int m_2;
};

int main (void)
{
    union my_union u;             /* Declare union */
    u.m_1 = 1;                    /* Write to m_1 */
    printf("u.m_2: %i\n", u.m_2); /* Read from m_2 */
    u.m_2 = 2;                    /* Write to m_2 */
    printf("u.m_1: %i\n", u.m_1); /* Read from m_1 */
    return 0;
}
```

Result

```
u.m_2: 1
u.m_1: 2
```

Section 56.3: Difference between struct and union

This illustrates that union members shares memory and that struct members does not share memory.

```
#include <stdio.h>
#include <string.h>

union My_Union
{
  int variable_1;
  int variable_2;
};

struct My_Struct
{
  int variable_1;
  int variable_2;
};

int main (void)
{
  union My_Union u;
  struct My_Struct s;
  u.variable_1 = 1;
  u.variable_2 = 2;
  s.variable_1 = 1;
  s.variable_2 = 2;
  printf ("u.variable_1: %i\n", u.variable_1);
  printf ("u.variable_2: %i\n", u.variable_2);
  printf ("s.variable_1: %i\n", s.variable_1);
  printf ("s.variable_2: %i\n", s.variable_2);
  printf ("sizeof (union My_Union): %i\n", sizeof (union My_Union));
  printf ("sizeof (struct My_Struct): %i\n", sizeof (struct My_Struct));
  return 0;
}
```

Chapter 57: Threads (native)

Section 57.1: Inititialization by one thread

In most cases all data that is accessed by several threads should be initialized before the threads are created. This ensures that all threads start with a clear state and no *race condition* occurs.

If this is not possible `once_flag` and `call_once` can be used

```
#include <threads.h>
#include <stdlib.h>

// the user data for this example
double const* Big = 0;

// the flag to protect big, must be global and/or static
static once_flag onceBig = ONCE_INIT;

void destroyBig(void) {
   free((void*)Big);
}

void initBig(void) {
    // assign to temporary with no const qualification
    double* b = malloc(largeNum);
    if (!b) {
       perror("allocation failed for Big");
       exit(EXIT_FAILURE);
    }
    // now initialize and store Big
    initializeBigWithSophisticatedValues(largeNum, b);
    Big = b;
    // ensure that the space is freed on exit or quick_exit
    atexit(destroyBig);
    at_quick_exit(destroyBig);
}

// the user thread function that relies on Big
int myThreadFunc(void* a) {
   call_once(&onceBig, initBig);
   // only use Big from here on
   ...
   return 0;
}
```

The `once_flag` is used to coordinate different threads that might want to initialize the same data `Big`. The call to `call_once` guarantees that

- `initBig` is called exactly once
- `call_once` blocks until such a call to `initBig` has been made, either by the same or another thread.

Besides allocation, a typical thing to do in such a once-called function is a dynamic initialization of a thread control data structures such as `mtx_t` or `cnd_t` that can't be initialized statically, using `mtx_init` or `cnd_init`, respectively.

Section 57.2: Start several threads

```
#include <stdio.h>
```

```c
#include <threads.h>
#include <stdlib.h>

struct my_thread_data {
   double factor;
};

int my_thread_func(void* a) {
   struct my_thread_data* d = a;
   // do something with d
   printf("we found %g\n", d->factor);
   // return an success or error code
   return d->factor > 1.0;
}


int main(int argc, char* argv[argc+1]) {
    unsigned n = 4;
    if (argc > 1) n = strtoull(argv[1], 0, 0);
    // reserve space for the arguments for the threads
    struct my_thread_data D[n];     // can't be initialized
    for (unsigned i = 0; i < n; ++i) {
         D[i] = (struct my_thread_data){ .factor = 0.5*i, };
    }
    // reserve space for the ID's of the threads
    thrd_t id[4];
    // launch the threads
    for (unsigned i = 0; i < n; ++i) {
         thrd_create(&id[i], my_thread_func, &D[i]);
    }
    // Wait that all threads have finished, but throw away their
    // return values
    for (unsigned i = 0; i < n; ++i) {
         thrd_join(id[i], 0);
    }
    return EXIT_SUCCESS;
}
```

Chapter 58: Multithreading

In C11 there is a standard thread library, **`<threads.h>`**, but no known compiler that yet implements it. Thus, to use multithreading in C you must use platform specific implementations such as the POSIX threads library (often referred to as pthreads) using the `pthread.h` header.

Section 58.1: C11 Threads simple example

```
#include <threads.h>
#include <stdio.h>

int run(void *arg)
{
    printf("Hello world of C11 threads.");

    return 0;
}

int main(int argc, const char *argv[])
{
    thrd_t thread;
    int result;

    thrd_create(&thread, run, NULL);

    thrd_join(&thread, &result);

    printf("Thread return %d at the end\n", result);
}
```

Chapter 59: Interprocess Communication (IPC)

Inter-process communication (IPC) mechanisms allow different independent processes to communicate with each other. Standard C does not provide any IPC mechanisms. Therefore, all such mechanisms are defined by the host operating system. POSIX defines an extensive set of IPC mechanisms; Windows defines another set; and other systems define their own variants.

Section 59.1: Semaphores

Semaphores are used to synchronize operations between two or more processes. POSIX defines two different sets of semaphore functions:

1. 'System V IPC' — `semctl()`, `semop()`, `semget()`.
2. 'POSIX Semaphores' — `sem_close()`, `sem_destroy()`, `sem_getvalue()`, `sem_init()`, `sem_open()`, `sem_post()`, `sem_trywait()`, `sem_unlink()`.

This section describes the System V IPC semaphores, so called because they originated with Unix System V.

First, you'll need to include the required headers. Old versions of POSIX required `#include <sys/types.h>`; modern POSIX and most systems do not require it.

```
#include <sys/sem.h>
```

Then, you'll need to define a key in both the parent as well as the child.

```
#define KEY 0x1111
```

This key needs to be the same in both programs or they will not refer to the same IPC structure. There are ways to generate an agreed key without hard-coding its value.

Next, depending on your compiler, you may or may not need to do this step: declare a union for the purpose of semaphore operations.

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};
```

Next, define your *try* (`semwait`) and *raise* (`semsignal`) structures. The names P and V originate from Dutch

```
struct sembuf p = { 0, -1, SEM_UNDO}; # semwait
struct sembuf v = { 0, +1, SEM_UNDO}; # semsignal
```

Now, start by getting the id for your IPC semaphore.

```
int id;
// 2nd argument is number of semaphores
// 3rd argument is the mode (IPC_CREAT creates the semaphore set if needed)
if ((id = semget(KEY, 1, 0666 | IPC_CREAT) < 0) {
    /* error handling code */
}
```

In the parent, initialise the semaphore to have a counter of 1.

```
union semun u;
u.val = 1;
if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL is a macro to specify that you're setting the value of
the semaphore to that specified by the union u
    /* error handling code */
}
```

Now, you can decrement or increment the semaphore as you need. At the start of your critical section, you decrement the counter using the `semop()` function:

```
if (semop(id, &p, 1) < 0) {
    /* error handling code */
}
```

To increment the semaphore, you use `&v` instead of `&p`:

```
if (semop(id, &v, 1) < 0) {
    /* error handling code */
}
```

Note that every function returns `0` on success and `-1` on failure. Not checking these return statuses can cause devastating problems.

Example 1.1: Racing with Threads

The below program will have a process `fork` a child and both parent and child attempt to print characters onto the terminal without any synchronization.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
```

```
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
}
```

Output (1st run):

```
aAABaBCbCbDDcEEcddeFFGGHHeffgghh
```

(2nd run):

```
aabbccAABddBCeeCffgDDghEEhFFGGHH
```

Compiling and running this program should give you a different output each time .

Example 1.2: Avoid Racing with Semaphores

Modifying *Example 1.1* to use semaphores, we have:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 0x1111

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};

struct sembuf p = { 0, -1, SEM_UNDO};
struct sembuf v = { 0, +1, SEM_UNDO};

int main()
{
    int id = semget(KEY, 1, 0666 | IPC_CREAT);
    if(id < 0)
    {
        perror("semget"); exit(11);
    }
    union semun u;
    u.val = 1;
    if(semctl(id, 0, SETVAL, u) < 0)
    {
```

```
        perror("semctl"); exit(12);
    }
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(13);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(14);
            }

            sleep(rand() % 2);
        }
    }
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(15);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(16);
            }

            sleep(rand() % 2);
        }
    }
}
```

Output:

```
aabbAABBCCccddeeDDffEEFFGGHHgghh
```

Compiling and running this program will give you the same output each time.

Chapter 60: Testing frameworks

Many developers use unit tests to check that their software works as expected. Unit tests check small units of larger pieces of software, and ensure that the outputs match expectations. Testing frameworks make unit testing easier by providing set-up/tear-down services and coordinating the tests.

There are many unit testing frameworks available for C. For example, Unity is a pure C framework. People quite often use C++ testing frameworks to test C code; there are many C++ test frameworks too.

Section 60.1: Unity Test Framework

Unity is an xUnit-style test framework for unit testing C. It is written completely in C and is portable, quick, simple, expressive and extensible. It is designed to especially be also useful for unit testing for embedded systems.

A simple test case that checks the return value of a function, might look as follows

```
void test_FunctionUnderTest_should_ReturnFive(void)
{
    TEST_ASSERT_EQUAL_INT( 5, FunctionUnderTest() );
}
```

A full test file might look like:

```
#include "unity.h"
#include "UnitUnderTest.h" /* The unit to be tested. */

void setUp (void) {} /* Is run before every test, put unit init calls here. */
void tearDown (void) {} /* Is run after every test, put unit clean-up calls here. */

void test_TheFirst(void)
{
    TEST_IGNORE_MESSAGE("Hello world!"); /* Ignore this test but print a message. */
}

int main (void)
{
    UNITY_BEGIN();
    RUN_TEST(test_TheFirst); /* Run the test. */
    return UNITY_END();
}
```

Unity comes with some example projects, makefiles and some Ruby rake scripts that help make creating longer test files a bit easier.

Section 60.2: CMocka

CMocka is an elegant unit testing framework for C with support for mock objects. It only requires the standard C library, works on a range of computing platforms (including embedded) and with different compilers. It has a tutorial on testing with mocks, API documentation, and a variety of examples.

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

void null_test_success (void ** state) {}
```

```c
void null_test_fail (void ** state)
{
    assert_true (0);
}

/* These functions will be used to initialize
   and clean resources up after each test run */
int setup (void ** state)
{
    return 0;
}

int teardown (void ** state)
{
    return 0;
}


int main (void)
{
    const struct CMUnitTest tests [] =
    {
        cmocka_unit_test (null_test_success),
        cmocka_unit_test (null_test_fail),
    };

    /* If setup and teardown functions are not
       needed, then NULL may be passed instead */

    int count_fail_tests =
        cmocka_run_group_tests (tests, setup, teardown);

    return count_fail_tests;
}
```

Section 60.3: CppUTest

CppUTest is an xUnit-style framework for unit testing C and C++. It is written in C++ and aims for portability and simplicity in design. It has support for memory leak detection, building mocks, and running its tests along with the Google Test. Comes with helper scripts and sample projects for Visual Studio and Eclipse CDT.

```c
#include <CppUTest/CommandLineTestRunner.h>
#include <CppUTest/TestHarness.h>


TEST_GROUP(Foo_Group) {}

TEST(Foo_Group, Foo_TestOne) {}

/* Test runner may be provided options, such
   as to enable colored output, to run only a
   specific test or a group of tests, etc. This
   will return the number of failed tests. */

int main(int argc, char ** argv)
{
    RUN_ALL_TESTS(argc, argv);
}
```

A test group may have a `setup()` and a `teardown()` method. The `setup` method is called prior to each test and the

`teardown()` method is called after. Both are optional and either may be omitted independently. Other methods and variables may also be declared inside a group and will be available to all tests of that group.

```
TEST_GROUP(Foo_Group)
{
    size_t data_bytes = 128;
    void * data;

    void setup()
    {
        data = malloc(data_bytes);
    }

    void teardown()
    {
        free(data);
    }

    void clear()
    {
        memset(data, 0, data_bytes);
    }
}
```

Chapter 61: Valgrind

Section 61.1: Bytes lost -- Forgetting to free

Here is a program that calls malloc but not free:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    char *s;

    s = malloc(26); // the culprint

    return 0;
}
```

With no extra arguments, valgrind will not look for this error.

But if we turn on `--leak-check=yes` or `--tool=memcheck`, it will complain and display the lines responsible for those memory leaks if the program was compiled in debug mode:

```
$ valgrind -q --leak-check=yes ./missing_free
==4776== 26 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4776==    at 0x4024F20: malloc (vg_replace_malloc.c:236)
==4776==    by 0x80483F8: main (missing_free.c:9)
==4776==
```

If the program is not compiled in debug mode (for example with the `-g` flag in GCC) it will still show us where the leak happened in terms of the relevant function, but not the lines.

This lets us go back and look at what block was allocated in that line and try to trace forward to see why it wasn't freed.

Section 61.2: Most common errors encountered while using Valgrind

Valgrind provides you with the *lines at which the error occurred* at the end of each line in the format `(file.c:line_no)`. Errors in valgrind are summarised in the following way:

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The most common errors include:

1. **Illegal read/write errors**

```
==8451== Invalid read of size 2
==8451==    at 0x4E7381D: getenv (getenv.c:84)
==8451==    by 0x4EB1559: __libc_message (libc_fatal.c:80)
==8451==    by 0x4F5256B: __fortify_fail (fortify_fail.c:37)
==8451==    by 0x4F5250F: __stack_chk_fail (stack_chk_fail.c:28)
==8451==    by 0x40059C: main (valg.c:10)
==8451==  Address 0x700000007 is not stack'd, malloc'd or (recently) free'd
```

This happens when the code starts to access memory which does not belong to the program. The size of the memory accessed also gives you an indication of what variable was used.

2. **Use of Uninitialized Variables**

```
==8795== 1 errors in context 5 of 8:
==8795== Conditional jump or move depends on uninitialised value(s)
==8795==    at 0x4E881AF: vfprintf (vfprintf.c:1631)
==8795==    by 0x4E8F898: printf (printf.c:33)
==8795==    by 0x400548: main (valg.c:7)
```

According to the error, at line 7 of the `main` of `valg.c`, the call to `printf()` passed an uninitialized variable to `printf`.

3. **Illegal freeing of Memory**

```
==8954== Invalid free() / delete / delete[] / realloc()
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x4005A8: main (valg.c:10)
==8954==  Address 0x5203040 is 0 bytes inside a block of size 240 free'd
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40059C: main (valg.c:9)
==8954==  Block was alloc'd at
==8954==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40058C: main (valg.c:7)
```

According to valgrind, the code freed the memory illegally (a second time) at *line 10* of `valg.c`, whereas it was already freed at *line 9*, and the block itself was allocated memory at *line 7*.

Section 61.3: Running Valgrind

```
valgrind ./my-program arg1 arg2 < test-input
```

This will run your program and produce a report of any allocations and de-allocations it did. It will also warn you about common errors like using uninitialized memory, dereferencing pointers to strange places, writing off the end of blocks allocated using malloc, or failing to free blocks.

Section 61.4: Adding flags

You can also turn on more tests, such as:

```
valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input
```

See valgrind --help for more information about the (many) options, or look at the documentation at http://valgrind.org/ for detailed information about what the output means.

Chapter 62: Common C programming idioms and developer practices

Section 62.1: Comparing literal and variable

Suppose you are comparing value with some variable

```
if ( i  == 2) //Bad-way
{
    doSomething;
}
```

Now suppose you have mistaken == with =. Then it will take your sweet time to figure it out.

```
if( 2 == i) //Good-way
{
    doSomething;
}
```

Then, if an equal sign is accidentally left out, the compiler will complain about an “attempted assignment to literal.” This won’t protect you when comparing two variables, but every little bit helps.

See here for more info.

Section 62.2: Do not leave the parameter list of a function blank — use void

Suppose you are creating a function that requires no arguments when it is called and you are faced with the dilemma of how you should define the parameter list in the function prototype and the function definition.

- You have the choice of keeping the parameter list empty for both prototype and definition. Thereby, they look just like the function call statement you will need.
- You read somewhere that one of the uses of keyword ***void*** (there are only a few of them), is to define the parameter list of functions that do not accept any arguments in their call. So, this is also a choice.

So, which is the correct choice?

ANSWER: using the keyword ***void***

GENERAL ADVICE: If a language provides certain feature to use for a special purpose, you are better off using that in your code. For example, using **`enum`**s instead of `#define` macros (that's for another example).

C11 section 6.7.6.3 "Function declarators", paragraph 10, states:

> The special case of an unnamed parameter of type void as the only item in the list specifies that the function has no parameters.

Paragraph 14 of that same section shows the only difference:

> … An empty list in a function declarator that is part of a definition of that function specifies that the

> function has no parameters. The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.

A simplified explanation provided by K&R (pgs- 72-73) for the above stuff:

> Furthermore, if a function declaration does not include arguments, as in `double atof();`, that too is taken to mean that nothing is to be assumed about the arguments of `atof`; all parameter checking is turned off. This special meaning of the empty argument list is intended to permit older C programs to compile with new compilers. But it's a bad idea to use it with new programs. If the function takes arguments, declare them; if it takes no arguments, use `void`.

So this is how your function prototype should look:

```
int foo(void);
```

And this is how the function definition should be:

```
int foo(void)
{
    ...
    <statements>
    ...
    return 1;
}
```

One advantage of using the above, over `int foo()` type of declaration (ie. without using the keyword ***void***), is that the compiler can detect the error if you call your function using an erroneous statement like `foo(42)`. This kind of a function call statement would not cause any errors if you leave the parameter list blank. The error would pass silently, undetected and the code would still execute.

This also means that you should define the `main()` function like this:

```
int main(void)
{
    ...
    <statements>
    ...
    return 0;
}
```

Note that even though a function defined with an empty parameter list takes no arguments, it does not provide a prototype for the function, so the compiler will not complain if the function is subsequently called with arguments. For example:

```
#include <stdio.h>

static void parameterless()
{
    printf("%s called\n", __func__);
}

int main(void)
{
    parameterless(3, "arguments", "provided");
```

```
    return 0;
}
```

If that code is saved in the file `proto79.c`, it can be compiled on Unix with GCC (version 7.1.0 on macOS Sierra 10.12.5 used for demonstration) like this:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o proto79
$
```

If you compile with more stringent options, you get errors:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-style-definition -pedantic proto79.c -o proto79
proto79.c:3:13: error: function declaration isn't a prototype [-Werror=strict-prototypes]
 static void parameterless()
             ^~~~~~~~~~~~~
proto79.c: In function 'parameterless':
proto79.c:3:13: error: old-style function definition [-Werror=old-style-definition]
cc1: all warnings being treated as errors
$
```

If you give the function the formal prototype `static void parameterless(void)`, then the compilation gives errors:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-style-definition -pedantic proto79.c -o proto79
proto79.c: In function 'main':
proto79.c:10:5: error: too many arguments to function 'parameterless'
     parameterless(3, "arguments", "provided");
     ^~~~~~~~~~~~~
proto79.c:3:13: note: declared here
 static void parameterless(void)
             ^~~~~~~~~~~~~
$
```

Moral — always make sure you have prototypes, and make sure your compiler tells you when you are not obeying the rules.

Chapter 63: Common pitfalls

This section discusses some of the common mistakes that a C programmer should be aware of and should avoid making. For more on some unexpected problems and their causes, please see Undefined behavior

Section 63.1: Mixing signed and unsigned integers in arithmetic operations

It is usually not a good idea to mix `signed` and `unsigned` integers in arithmetic operations. For example, what will be output of following example?

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1000;
    signed int b = -1;

    if (a > b) puts("a is more than b");
    else puts("a is less or equal than b");

    return 0;
}
```

Since 1000 is more than -1 you would expect the output to be `a is more than b`, however that will not be the case.

Arithmetic operations between different integral types are performed within a common type defined by the so called usual arithmetic conversions (see the language specification, 6.3.1.8).

In this case the "common type" is `unsigned int`, Because, as stated in Usual arithmetic conversions,

> 714 Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

This means that `int` operand `b` will get converted to `unsigned int` before the comparison.

When -1 is converted to an `unsigned int` the result is the maximal possible `unsigned int` value, which is greater than 1000, meaning that `a > b` is false.

Section 63.2: Macros are simple string replacements

Macros are simple string replacements. (Strictly speaking, they work with preprocessing tokens, not arbitrary strings.)

```
#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

You may expect this code to print `9` (`3*3`), but actually `5` will be printed because the macro will be expanded to `1+2*1+2`.

You should wrap the arguments and the whole macro expression in parentheses to avoid this problem.

```
#include <stdio.h>

#define SQUARE(x) ((x)*(x))

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Another problem is that the arguments of a macro are not guaranteed to be evaluated once; they may not be evaluated at all, or may be evaluated multiple times.

```
#include <stdio.h>

#define MIN(x, y) ((x) <= (y) ? (x) : (y))

int main(void) {
    int a = 0;
    printf("%d\n", MIN(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

In this code, the macro will be expanded to `((a++) <= (10) ? (a++) : (10))`. Since `a++` (`0`) is smaller than `10`, `a++` will be evaluated twice and it will make the value of `a` and what is returned from `MIN` differ from you may expect.

This can be avoided by using functions, but note that the types will be fixed by the function definition, whereas macros can be (too) flexible with types.

```
#include <stdio.h>

int min(int x, int y) {
    return x <= y ? x : y;
}

int main(void) {
    int a = 0;
    printf("%d\n", min(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

Now the problem of double-evaluation is fixed, but this `min` function cannot deal with `double` data without truncating, for example.

Macro directives can be of two types:

```
#define OBJECT_LIKE_MACRO     followed by a "replacement list" of preprocessor tokens
#define FUNCTION_LIKE_MACRO(with, arguments) followed by a replacement list
```

What distinguishes these two types of macros is the character that follows the identifier after `#define`: if it's an *lparen*, it is a function-like macro; otherwise, it's an object-like macro. If the intention is to write a function-like

macro, there must not be any white space between the end of the name of the macro and `(`. Check this for a detailed explanation.

Version ≥ C99

In C99 or later, you could use `static inline int min(int x, int y) { … }`.

Version ≥ C11

In C11, you could write a 'type-generic' expression for `min`.

```
#include <stdio.h>

#define min(x, y) _Generic((x), \
                        long double: min_ld, \
                        unsigned long long: min_ull, \
                        default: min_i \
                        )(x, y)

#define gen_min(suffix, type) \
    static inline type min_##suffix(type x, type y) { return (x < y) ? x : y; }

gen_min(ld, long double)
gen_min(ull, unsigned long long)
gen_min(i, int)

int main(void)
{
    unsigned long long ull1 = 50ULL;
    unsigned long long ull2 = 37ULL;
    printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));
    long double ld1 = 3.141592653L;
    long double ld2 = 3.141592652L;
    printf("min(%.10Lf, %.10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));
    int i1 = 3141653;
    int i2 = 3141652;
    printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));
    return 0;
}
```

The generic expression could be extended with more types such as `double`, `float`, `long long`, `unsigned long`, `long`, `unsigned` — and appropriate `gen_min` macro invocations written.

Section 63.3: Forgetting to copy the return value of realloc into a temporary

If `realloc` fails, it returns `NULL`. If you assign the value of the original buffer to `realloc`'s return value, and if it returns `NULL`, then the original buffer (the old pointer) is lost, resulting in a *memory leak*. The solution is to copy into a temporary pointer, and if that temporary is not NULL, **then** copy into the real buffer.

```
char *buf, *tmp;

buf = malloc(...);
...

/* WRONG */
if ((buf = realloc(buf, 16)) == NULL)
    perror("realloc");
```

```
/* RIGHT */
if ((tmp = realloc(buf, 16)) != NULL)
    buf = tmp;
else
    perror("realloc");
```

Section 63.4: Forgetting to allocate one extra byte for \0

When you are copying a string into a `malloc`ed buffer, always remember to add 1 to `strlen`.

```
char *dest = malloc(strlen(src)); /* WRONG */
char *dest = malloc(strlen(src) + 1); /* RIGHT */

strcpy(dest, src);
```

This is because `strlen` does not include the trailing `\0` in the length. If you take the `WRONG` (as shown above) approach, upon calling `strcpy`, your program would invoke undefined behaviour.

It also applies to situations when you are reading a string of known maximum length from `stdin` or some other source. For example

```
#define MAX_INPUT_LEN 42

char buffer[MAX_INPUT_LEN]; /* WRONG */
char buffer[MAX_INPUT_LEN + 1]; /* RIGHT */

scanf("%42s", buffer);  /* Ensure that the buffer is not overflowed */
```

Section 63.5: Misunderstanding array decay

A common problem in code that uses multidimensional arrays, arrays of pointers, etc. is the fact that `Type**` and `Type[M][N]` are fundamentally different types:

```
#include <stdio.h>

void print_strings(char **strings, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(s, 4);
    return 0;
}
```

Sample compiler output:

```
file1.c: In function 'main':
file1.c:13:23: error: passing argument 1 of 'print_strings' from incompatible pointer type [-
Wincompatible-pointer-types]
         print_strings(strings, 4);
                       ^
file1.c:3:10: note: expected 'char **' but argument is of type 'char (*)[20]'
```

```
void print_strings(char **strings, size_t n)
```

The error states that the `s` array in the `main` function is passed to the function `print_strings`, which expects a different pointer type than it received. It also includes a note expressing the type that is expected by `print_strings` and the type that was passed to it from `main`.

The problem is due to something called *array decay*. What happens when `s` with its type `char[4][20]` (array of 4 arrays of 20 chars) is passed to the function is it turns into a pointer to its first element as if you had written `&s[0]`, which has the type `char (*)[20]` (pointer to 1 array of 20 chars). This occurs for any array, including an array of pointers, an array of arrays of arrays (3-D arrays), and an array of pointers to an array. Below is a table illustrating what happens when an array decays. Changes in the type description are highlighted to illustrate what happens:

Before Decay		After Decay	
`char [20]`	**array of (20 chars)**	`char *`	**pointer to (1 char)**
`char [4][20]`	**array of (4 arrays** of 20 chars)	`char (*)[20]`	**pointer to (1 array** of 20 chars)
`char *[4]`	**array of (4 pointers** to 1 char)	`char **`	**pointer to (1 pointer** to 1 char)
`char [3][4][20]`	**array of (3 arrays** of 4 arrays of 20 chars)	`char (*)[4][20]`	**pointer to (1 array** of 4 arrays of 20 chars)
`char (*[4])[20]`	**array of (4 pointers** to 1 array of 20 chars)	`char (**)[20]`	**pointer to (1 pointer** to 1 array of 20 chars)

If an array can decay to a pointer, then it can be said that a pointer may be considered an array of at least 1 element. An exception to this is a null pointer, which points to nothing and is consequently not an array.

Array decay only happens once. If an array has decayed to a pointer, it is now a pointer, not an array. Even if you have a pointer to an array, remember that the pointer might be considered an array of at least one element, so array decay has already occurred.

In other words, a pointer to an array (`char (*)[20]`) will never become a pointer to a pointer (`char **`). To fix the `print_strings` function, simply make it receive the correct type:

```
void print_strings(char (*strings)[20], size_t n)
/* OR */
void print_strings(char strings[][20], size_t n)
```

A problem arises when you want the `print_strings` function to be generic for any array of chars: what if there are 30 chars instead of 20? Or 50? The answer is to add another parameter before the array parameter:

```
#include <stdio.h>

/*
 * Note the rearranged parameters and the change in the parameter name
 * from the previous definitions:
 *      n (number of strings)
 *   => scount (string count)
 *
 * Of course, you could also use one of the following highly recommended forms
 * for the `strings` parameter instead:
 *
 *    char strings[scount][ccount]
 *    char strings[][ccount]
 */
void print_strings(size_t scount, size_t ccount, char (*strings)[ccount])
{
    size_t i;
    for (i = 0; i < scount; i++)
```

```
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(4, 20, s);
    return 0;
}
```

Compiling it produces no errors and results in the expected output:

```
Example 1
Example 2
Example 3
Example 4
```

Section 63.6: Forgetting to free memory (memory leaks)

A programming best practice is to free any memory that has been allocated directly by your own code, or implicitly by calling an internal or external function, such as a library API like `strdup()`. Failing to free memory can introduce a memory leak, which could accumulate into a substantial amount of wasted memory that is unavailable to your program (or the system), possibly leading to crashes or undefined behavior. Problems are more likely to occur if the leak is incurred repeatedly in a loop or recursive function. The risk of program failure increases the longer a leaking program runs. Sometimes problems appear instantly; other times problems won't be seen for hours or even years of constant operation. Memory exhaustion failures can be catastrophic, depending on the circumstances.

The following infinite loop is an example of a leak that will eventually exhaust available memory leak by calling `getline()`, a function that implicitly allocates new memory, without freeing that memory.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    /* The loop below leaks memory as fast as it can */

    for(;;) {
        getline(&line, &size, stdin); /* New memory implicitly allocated */

        /* <do whatever> */

        line = NULL;
    }

    return 0;
}
```

In contrast, the code below also uses the `getline()` function, but this time, the allocated memory is correctly freed, avoiding a leak.

```
#include <stdlib.h>
#include <stdio.h>
```

```c
int main(void)
{
    char *line = NULL;
    size_t size = 0;

    for(;;) {
        if (getline(&line, &size, stdin) < 0) {
            free(line);
            line = NULL;

            /* Handle failure such as setting flag, breaking out of loop and/or exiting */
        }

        /* <do whatever> */

        free(line);
        line = NULL;

    }

    return 0;
}
```

Leaking memory doesn't always have tangible consequences and isn't necessarily a functional problem. While "best practice" dictates rigorously freeing memory at strategic points and conditions, to reduce memory footprint and lower risk of memory exhaustion, there can be exceptions. For example, if a program is bounded in duration and scope, the risk of allocation failure might be considered too small to worry about. In that case, bypassing explicit deallocation might be considered acceptable. For example, most modern operating systems automatically free all memory consumed by a program when it terminates, whether it is due to program failure, a system call to `exit()`, process termination, or reaching end of `main()`. Explicitly freeing memory at the point of imminent program termination could actually be redundant or introduce a performance penalty.

Allocation can fail if insufficient memory is available, and handling failures should be accounted for at appropriate levels of the call stack. `getline()`, shown above is an interesting use-case because it is a library function that not only allocates memory it leaves to the caller to free, but can fail for a number of reasons, all of which must be taken into account. Therefore, it is essential when using a C API, to read the *documentation (man page)* and pay particular attention to error conditions and memory usage, and be aware which software layer bears the burden of freeing returned memory.

Another common memory handling practice is to consistently set memory pointers to NULL immediately after the memory referenced by those pointers is freed, so those pointers can be tested for validity at any time (e.g. checked for NULL / non-NULL), because accessing freed memory can lead to severe problems such as getting garbage data (read operation), or data corruption (write operation) and/or a program crash. In most modern operating systems, freeing memory location 0 (`NULL`) is a NOP (e.g. it is harmless), as required by the C standard — so by setting a pointer to NULL, there is no risk of double-freeing memory if the pointer is passed to `free()`. Keep in mind that double-freeing memory can lead to very time consuming, confusing, and *difficult to diagnose* failures.

Section 63.7: Copying too much

```c
char buf[8]; /* tiny buffer, easy to overflow */

printf("What is your name?\n");
scanf("%s", buf); /* WRONG */
scanf("%7s", buf); /* RIGHT */
```

If the user enters a string longer than 7 characters (- 1 for the null terminator), memory behind the buffer `buf` will

be overwritten. This results in undefined behavior. Malicious hackers often exploit this in order to overwrite the return address, and change it to the address of the hacker's malicious code.

Section 63.8: Mistakenly writing = instead of == when comparing

The `=` operator is used for assignment.

The `==` operator is used for comparison.

One should be careful not to mix the two. Sometimes one mistakenly writes

```
/* assign y to x */
if (x = y) {
     /* logic */
}
```

when what was really wanted is:

```
/* compare if x is equal to y */
if (x == y) {
    /* logic */
}
```

The former assigns value of y to x and checks if that value is non zero, instead of doing comparison, which is equivalent to:

```
if ((x = y) != 0) {
    /* logic */
}
```

There are times when testing the result of an assignment is intended and is commonly used, because it avoids having to duplicate code and having to treat the first time specially. Compare

```
while ((c = getopt_long(argc, argv, short_options, long_options, &option_index)) != -1) {
        switch (c) {
        ...
        }
}
```

versus

```
c = getopt_long(argc, argv, short_options, long_options, &option_index);
while (c != -1) {
        switch (c) {
        ...
        }
        c = getopt_long(argc, argv, short_options, long_options, &option_index);
}
```

Modern compilers will recognise this pattern and do not warn when the assignment is inside parenthesis like above, but may warn for other usages. For example:

```
if (x = y)         /* warning */

if ((x = y))       /* no warning */
```

```
if ((x = y) != 0)  /* no warning; explicit */
```

Some programmers use the strategy of putting the constant to the left of the operator (commonly called Yoda conditions). Because constants are rvalues, this style of condition will cause the compiler to throw an error if the wrong operator was used.

```
if (5 = y) /* Error */

if (5 == y) /* No error */
```

However, this severely reduces the readability of the code and is not considered necessary if the programmer follows good C coding practices, and doesn't help when comparing two variables so it isn't a universal solution. Furthermore, many modern compilers may give warnings when code is written with Yoda conditions.

Section 63.9: Newline character is not consumed in typical scanf() call

When this program

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;

    scanf("%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

is executed with this input

```
42
life
```

the output will be `42 ""` instead of expected `42 "life"`.

This is because a newline character after 42 is not consumed in the call of `scanf()` and it is consumed by `fgets()` before it reads `life`. Then, `fgets()` stop reading before reading `life`.

To avoid this problem, one way that is useful when the maximum length of a line is known -- when solving problems in online judge syste, for example -- is avoiding using `scanf()` directly and reading all lines via `fgets()`. You can use `sscanf()` to parse the lines read.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char line_buffer[128] = "", str[128], *lf;

    fgets(line_buffer, sizeof(line_buffer), stdin);
    sscanf(line_buffer, "%d", &num);
```

```
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Another way is to read until you hit a newline character after using `scanf()` and before using `fgets()`.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;
    int c;

    scanf("%d", &num);
    while ((c = getchar()) != '\n' && c != EOF);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Section 63.10: Adding a semicolon to a #define

It is easy to get confused in the C preprocessor, and treat it as part of C itself, but that is a mistake because the preprocessor is just a text substitution mechanism. For example, if you write

```
/* WRONG */
#define MAX 100;
int arr[MAX];
```

the code expands to

```
int arr[100;];
```

which is a syntax error. The remedy is to remove the semicolon from the `#define` line. It is almost invariably a mistake to end a `#define` with a semicolon.

Section 63.11: Incautious use of semicolons

Be careful with semicolons. Following example

```
if (x > a);
   a = x;
```

actually means:

```
if (x > a) {}
a = x;
```

which means `x` will be assigned to `a` in any case, which might not be what you wanted originally.

Sometimes, missing a semicolon will also cause an unnoticeable problem:

```
if (i < 0)
    return
day = date[0];
hour = date[1];
minute = date[2];
```

The semicolon behind return is missed, so day=date[0] will be returned.

One technique to avoid this and similar problems is to always use braces on multi-line conditionals and loops. For example:

```
if (x > a) {
    a = x;
}
```

Section 63.12: Undefined reference errors when linking

One of the most common errors in compilation happens during the linking stage. The error looks similar to this:

```
$ gcc undefined_reference.c
/tmp/ccoXhwF0.o: In function `main':
undefined_reference.c:(.text+0x15): undefined reference to `foo'
collect2: error: ld returned 1 exit status
$
```

So let's look at the code that generated this error:

```
int foo(void);

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

We see here a *declaration* of foo (`int foo();`) but no *definition* of it (actual function). So we provided the compiler with the function header, but there was no such function defined anywhere, so the compilation stage passes but the linker exits with an `Undefined reference` error.
To fix this error in our small program we would only have to add a *definition* for foo:

```
/* Declaration of foo */
int foo(void);

/* Definition of foo */
int foo(void)
{
    return 5;
}

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Now this code will compile. An alternative situation arises where the source for `foo()` is in a separate source file `foo.c` (and there's a header `foo.h` to declare `foo()` that is included in both `foo.c` and `undefined_reference.c`). Then the fix is to link both the object file from `foo.c` and `undefined_reference.c`, or to compile both the source files:

```
$ gcc -c undefined_reference.c
$ gcc -c foo.c
$ gcc -o working_program undefined_reference.o foo.o
$
```

Or:

```
$ gcc -o working_program undefined_reference.c foo.c
$
```

A more complex case is where libraries are involved, like in the code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
    double first;
    double second;
    double power;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <denom> <nom>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Translate user input to numbers, extra error checking
     * should be done here. */
    first = strtod(argv[1], NULL);
    second = strtod(argv[2], NULL);

    /* Use function pow() from libm - this will cause a linkage
     * error unless this code is compiled against libm! */
    power = pow(first, second);

    printf("%f to the power of %f = %f\n", first, second, power);

    return EXIT_SUCCESS;
}
```

The code is syntactically correct, declaration for `pow()` exists from `#include <math.h>`, so we try to compile and link but get an error like this:

```
$ gcc no_library_in_link.c -o no_library_in_link
/tmp/ccduQQqA.o: In function `main':
no_library_in_link.c:(.text+0x8b): undefined reference to `pow'
collect2: error: ld returned 1 exit status
$
```

This happens because the *definition* for `pow()` wasn't found during the linking stage. To fix this we have to specify we want to link against the math library called `libm` by specifying the `-lm` flag. (Note that there are platforms such

as macOS where `-lm` is not needed, but when you get the undefined reference, the library is needed.)

So we run the compilation stage again, this time specifying the library (after the source or object files):

```
$ gcc no_library_in_link.c -lm -o library_in_link_cmd
$ ./library_in_link_cmd 2 4
2.000000 to the power of 4.000000 = 16.000000
$
```

And it works!

Section 63.13: Checking logical expression against 'true'

The original C standard had no intrinsic Boolean type, so `bool`, **`true`** and **`false`** had no inherent meaning and were often defined by programmers. Typically **`true`** would be defined as 1 and **`false`** would be defined as 0.

Version ≥ C99

C99 adds the built-in type `_Bool` and the header **`<stdbool.h>`** which defines `bool` (expanding to `_Bool`), **`false`** and **`true`**. It also allows you to redefine `bool`, **`true`** and **`false`**, but notes that this is an obsolescent feature.

More importantly, logical expressions treat anything that evaluates to zero as false and any non-zero evaluation as true. For example:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == true)  /* Comparison only succeeds if true is 0x80 and bitField has
that bit set */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

In the above example, the function is trying to check if the upper bit is set and return **`true`** if it is. However, by explicitly checking against **`true`**, the `if` statement will only succeed if `(bitfield & 0x80)` evaluates to whatever **`true`** is defined as, which is typically `1` and very seldom `0x80`. Either explicitly check against the case you expect:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == 0x80) /* Explicitly test for the case we expect */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Or evaluate any non-zero value as true.

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    /* If upper bit is set, result is 0x80 which the if will evaluate as true */
    if (bitField & 0x80)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Section 63.14: Doing extra scaling in pointer arithmetic

In pointer arithmetic, the integer to be added or subtracted to pointer is interpreted not as change of *address* but as number of *elements* to move.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + sizeof(int) * 2; /* wrong */
    printf("%d %d\n", *ptr, *ptr2);
    return 0;
}
```

This code does extra scaling in calculating pointer assigned to `ptr2`. If `sizeof(int)` is 4, which is typical in modern 32-bit environments, the expression stands for "8 elements after `array[0]`", which is out-of-range, and it invokes *undefined behavior*.

To have `ptr2` point at what is 2 elements after `array[0]`, you should simply add 2.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + 2;
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

Explicit pointer arithmetic using additive operators may be confusing, so using array subscripting may be better.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = &ptr[2];
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

`E1[E2]` is identical to `(*((E1)+(E2)))` (N1570 6.5.2.1, paragraph 2), and `&(E1[E2])` is equivalent to `((E1)+(E2))`

(N1570 6.5.3.2, footnote 102).

Alternatively, if pointer arithmetic is preferred, casting the pointer to address a different data type can allow byte addressing. Be careful though: endianness can become an issue, and casting to types other than 'pointer to character' leads to strict aliasing problems.

```
#include <stdio.h>

int main(void) {
    int array[3] = {1,2,3};  // 4 bytes * 3 allocated
    unsigned char *ptr = (unsigned char *) array;  // unsigned chars only take 1 byte
    /*
     * Now any pointer arithmetic on ptr will match
     * bytes in memory.  ptr can be treated like it
     * was declared as: unsigned char ptr[12];
     */

    return 0;
}
```

Section 63.15: Multi-line comments cannot be nested

In C, multi-line comments, /* and */, do not nest.

If you annotate a block of code or function using this style of comment:

```
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

You will not be able to comment it out easily:

```
//Trying to comment out the block...
/*

/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
```

```
}

//Causes an error on the line below...
*/
```

One solution is to use C99 style comments:

```
// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Now the entire block can be commented out easily:

```
/*

// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

*/
```

Another solution is to avoid disabling code using comment syntax, using `#ifdef` or `#ifndef` preprocessor directives instead. These directives *do* nest, leaving you free to comment your code in the style you prefer.

```
#define DISABLE_MAX /* Remove or comment this line to enable max() code block */

#ifdef DISABLE_MAX
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

```
#endif
```

Some guides go so far as to recommend that code sections must *never* be commented and that if code is to be temporarily disabled one could resort to using an `#if 0` directive.

See #if 0 to block out code sections.

Section 63.16: Ignoring return values of library functions

Almost every function in C standard library returns something on success, and something else on error. For example, `malloc` will return a pointer to the memory block allocated by the function on success, and, if the function failed to allocate the requested block of memory, a null pointer. So you should always check the return value for easier debugging.

This is bad:

```
char* x = malloc(100000000000UL * sizeof *x);
/* more code */
scanf("%s", x); /* This might invoke undefined behaviour and if lucky causes a segmentation
violation, unless your system has a lot of memory */
```

This is good:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char* x = malloc(100000000000UL * sizeof *x);
    if (x == NULL) {
        perror("malloc() failed");
        exit(EXIT_FAILURE);
    }

    if (scanf("%s", x) != 1) {
        fprintf(stderr, "could not read string\n");
        free(x);
        exit(EXIT_FAILURE);
    }

    /* Do stuff with x. */

    /* Clean up. */
    free(x);

    return EXIT_SUCCESS;
}
```

This way you know right away the cause of error, otherwise you might spend hours looking for a bug in a completely wrong place.

Section 63.17: Comparing floating point numbers

Floating point types (`float`, `double` and `long double`) cannot precisely represent some numbers because they have finite precision and represent the values in a binary format. Just like we have repeating decimals in base 10 for fractions such as 1/3, there are fractions that cannot be represented finitely in binary too (such as 1/3, but also, more importantly, 1/10). Do not directly compare floating point values; use a delta instead.

```c
#include <float.h> // for DBL_EPSILON and FLT_EPSILON
#include <math.h>  // for fabs()

int main(void)
{
    double a = 0.1; // imprecise: (binary) 0.000110...

    // may be false or true
    if (a + a + a + a + a + a + a + a + a + a == 1.0) {
        printf("10 * 0.1 is indeed 1.0. This is not guaranteed in the general case.\n");
    }

    // Using a small delta value.
    if (fabs(a + a + a + a + a + a + a + a + a + a - 1.0) < 0.000001) {
        // C99 5.2.4.2.2p8 guarantees at least 10 decimal digits
        // of precision for the double type.
        printf("10 * 0.1 is almost 1.0.\n");
    }

    return 0;
}
```

Another example:

```c
gcc -O3    -g    -I./inc    -std=c11    -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes
 -Wold-style-definition       rd11.c -o rd11 -L./lib -lsoq
#include <stdio.h>
#include <math.h>

static inline double rel_diff(double a, double b)
{
    return fabs(a - b) / fmax(fabs(a), fabs(b));
}

int main(void)
{
    double d1 = 3.14159265358979;
    double d2 = 355.0 / 113.0;

    double epsilon = 1.0;
    for (int i = 0; i < 10; i++)
    {
        if (rel_diff(d1, d2) < epsilon)
            printf("%d:%.10f <=> %.10f within tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        else
            printf("%d:%.10f <=> %.10f out of tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        epsilon /= 10.0;
    }
    return 0;
}
```

Output:

```
0:3.1415926536 <=> 3.1415929204 within tolerance 1.0000000000 (rel diff 8.4914E-08)
1:3.1415926536 <=> 3.1415929204 within tolerance 0.1000000000 (rel diff 8.4914E-08)
2:3.1415926536 <=> 3.1415929204 within tolerance 0.0100000000 (rel diff 8.4914E-08)
3:3.1415926536 <=> 3.1415929204 within tolerance 0.0010000000 (rel diff 8.4914E-08)
4:3.1415926536 <=> 3.1415929204 within tolerance 0.0001000000 (rel diff 8.4914E-08)
```

```
5:3.1415926536 <=> 3.1415929204 within tolerance 0.0000100000 (rel diff 8.4914E-08)
6:3.1415926536 <=> 3.1415929204 within tolerance 0.0000010000 (rel diff 8.4914E-08)
7:3.1415926536 <=> 3.1415929204 within tolerance 0.0000001000 (rel diff 8.4914E-08)
8:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000100 (rel diff 8.4914E-08)
9:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000010 (rel diff 8.4914E-08)
```

Section 63.18: Floating point literals are of type double by default

Care must be taken when initializing variables of type `float` to literal values or comparing them with literal values, because regular floating point literals like `0.1` are of type `double`. This may lead to surprises:

```
#include <stdio.h>
int main() {
    float  n;
    n = 0.1;
    if (n > 0.1) printf("Wierd\n");
    return 0;
}
// Prints "Wierd" when n is float
```

Here, `n` gets initialized and rounded to single precision, resulting in value 0.10000000149011612. Then, `n` is converted back to double precision to be compared with `0.1` literal (which equals to 0.10000000000000001), resulting in a mismatch.

Besides rounding errors, mixing `float` variables with `double` literals will result in poor performance on platforms which don't have hardware support for double precision.

Section 63.19: Using character constants instead of string literals, and vice versa

In C, character constants and string literals are different things.

A character surrounded by single quotes like `'a'` is a *character constant*. A character constant is an integer whose value is the character code that stands for the character. How to interpret character constants with multiple characters like `'abc'` is implementation-defined.

Zero or more characters surrounded by double quotes like `"abc"` is a *string literal*. A string literal is an unmodifiable array whose elements are type `char`. The string in the double quotes plus terminating null-character are the contents, so `"abc"` has 4 elements (`{'a', 'b', 'c', '\0'}`)

In this example, a character constant is used where a string literal should be used. This character constant will be converted to a pointer in an implementation-defined manner and there is little chance for the converted pointer to be valid, so this example will invoke *undefined behavior*.

```
#include <stdio.h>

int main(void) {
    const char *hello = 'hello, world'; /* bad */
    puts(hello);
    return 0;
}
```

In this example, a string literal is used where a character constant should be used. The pointer converted from the string literal will be converted to an integer in an implementation-defined manner, and it will be converted to `char`

in an implementation-defined manner. (How to convert an integer to a signed type which cannot represent the value to convert is implementation-defined, and whether `char` is signed is also implementation-defined.) The output will be some meaningless thing.

```
#include <stdio.h>

int main(void) {
    char c = "a"; /* bad */
    printf("%c\n", c);
    return 0;
}
```

In almost all cases, the compiler will complain about these mix-ups. If it doesn't, you need to use more compiler warning options, or it is recommended that you use a better compiler.

Section 63.20: Recursive function — missing out the base condition

Calculating the factorial of a number is a classic example of a recursive function.

Missing the Base Condition:

```
#include <stdio.h>

int factorial(int n)
{
       return n * factorial(n - 1);
}

int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Typical output: `Segmentation fault: 11`

The problem with this function is it would loop infinitely, causing a segmentation fault — it needs a base condition to stop the recursion.

Base Condition Declared:

```
#include <stdio.h>

int factorial(int n)
{
    if (n == 1) // Base Condition, very crucial in designing the recursive functions.
    {
       return 1;
    }
    else
    {
       return n * factorial(n - 1);
    }
}

int main()
{
```

```
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Sample output

```
Factorial 3 = 6
```

This function will terminate as soon as it hits the condition `n` is equal to 1 (provided the initial value of `n` is small enough — the upper bound is `12` when `int` is a 32-bit quantity).

Rules to be followed:

1. Initialize the algorithm. Recursive programs often need a seed value to start with. This is accomplished either by using a parameter passed to the function or by providing a gateway function that is non-recursive but that sets up the seed values for the recursive calculation.
2. Check to see whether the current value(s) being processed match the base case. If so, process and return the value.
3. Redefine the answer in terms of a smaller or simpler sub-problem or sub-problems.
4. Run the algorithm on the sub-problem.
5. Combine the results in the formulation of the answer.
6. Return the results.

Source: [Recursive Function]

Section 63.21: Overstepping array boundaries

Arrays are zero-based, that is the index always starts at 0 and ends with index array length minus 1. Thus the following code will not output the first element of the array and will output garbage for the final value that it prints.

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 1; x <= 5; x++) //Looping from 1 till 5.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Output: `2 3 4 5 GarbageValue`

The following demonstrates the correct way to achieve the desired output:

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 0; x < 5; x++) //Looping from 0 till 4.
       printf("%d\t", myArray[x]);
```

```
    printf("\n");
    return 0;
}
```

Output: `1 2 3 4 5`

It is important to know the length of an array before working with it as otherwise you may corrupt the buffer or cause a segmentation fault by accessing memory locations that are out of bounds.

Section 63.22: Passing unadjacent arrays to functions expecting "real" multidimensional arrays

When allocating multidimensional arrays with `malloc`, `calloc`, and `realloc`, a common pattern is to allocate the inner arrays with multiple calls (even if the call only appears once, it may be in a loop):

```
/* Could also be `int **` with malloc used to allocate outer array. */
int *array[4];
int i;

/* Allocate 4 arrays of 16 ints. */
for (i = 0; i < 4; i++)
    array[i] = malloc(16 * sizeof(*array[i]));
```

The difference in bytes between the last element of one of the inner arrays and the first element of the next inner array may not be 0 as they would be with a "real" multidimensional array (e.g. `int array[4][16];`):

```
/* 0x40003c, 0x402000 */
printf("%p, %p\n", (void *)(array[0] + 15), (void *)array[1]);
```

Taking into account the size of `int`, you get a difference of 8128 bytes (8132-4), which is 2032 `int`-sized array elements, and that is the problem: a "real" multidimensional array has no gaps between elements.

If you need to use a dynamically allocated array with a function expecting a "real" multidimensional array, you should allocate an object of type `int *` and use arithmetic to perform calculations:

```
void func(int M, int N, int *array);
...

/* Equivalent to declaring `int array[M][N] = {{0}};` and assigning to array4_16[i][j]. */
int *array;
int M = 4, N = 16;
array = calloc(M, N * sizeof(*array));
array[i * N + j] = 1;
func(M, N, array);
```

If `N` is a macro or an integer literal rather than a variable, the code can simply use the more natural 2-D array notation after allocating a pointer to an array:

```
void func(int M, int N, int *array);
#define N 16
void func_N(int M, int (*array)[N]);
...

int M = 4;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;
```

```c
/* Cast to `int *` works here because `array` is a single block of M*N ints with no gaps,
   just like `int array2[M * N];` and `int array3[M][N];` would be. */
func(M, N, (int *)array);
func_N(M, array);
```

Version ≥ C99

If `N` is not a macro or an integer literal, then `array` will point to a variable-length array (VLA). This can still be used with `func` by casting to `int *` and a new function `func_vla` would replace `func_N`:

```c
void func(int M, int N, int *array);
void func_vla(int M, int N, int array[M][N]);
...

int M = 4, N = 16;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;
func(M, N, (int *)array);
func_vla(M, N, array);
```

Version ≥ C11

Note: VLAs are optional as of C11. If your implementation supports C11 and defines the macro `__STDC_NO_VLA__` to 1, you are stuck with the pre-C99 methods.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

2501	Chapters 23, 28, 33, 34 and 36
3442	Chapter 4
4386427	Chapters 11, 19 and 46
A B	Chapter 19
abacles	Chapters 61 and 63
Alejandro Caro	Chapter 12
Aleksi Torhamo	Chapter 28
Alex	Chapter 55
Alex Garcia	Chapter 30
alk	Chapters 1, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 15, 20, 21, 24, 25, 28, 30, 37, 40, 43, 44, 46, 49 and 63
Altece	Chapter 22
Amani Kilumanga	Chapter 4
AnArrayOfFunctions	Chapters 10 and 44
Andrea Corbelli	Chapter 4
Andrey Markeev	Chapters 6 and 46
Ankush	Chapters 1, 2, 46 and 61
Antti Haapala	Chapters 22, 28 and 46
Armali	Chapters 23, 28 and 54
ArturFH	Chapter 1
AShelly	Chapters 10 and 33
Bakhtiar Hasan	Chapter 4
Ben Steffan	Chapter 28
BenG	Chapter 4
bevenson	Chapters 6, 11, 16, 20, 30, 37, 38 and 63
Blacksilver	Chapters 37 and 44
Blagovest Buyukliev	Chapters 3, 14, 33, 41 and 43
blatinox	Chapter 4
Bob__	Chapter 5
Braden Best	Chapter 5
bta	Chapters 22, 28 and 33
BurnsBA	Chapter 28
Buser	Chapter 42
catalogue_number	Chapter 6
cdrini	Chapter 10
Chandrahas Aroori	Chapters 2, 42 and 61
chqrlie	Chapter 1
Chris Sprague	Chapter 6
Christoph	Chapter 28
Chrono Kitsune	Chapters 6, 13, 43 and 63
Cimbali	Chapter 35
clearlight	Chapter 63
Cody Gray	Chapter 6
cshu	Chapter 28
cSmout	Chapter 10
c□ʟ□s□□□□	Chapter 59
DaBler	Chapter 28

Daksh Gupta	Chapter 46
Damien	Chapters 4 and 6
Daniel	Chapter 6
Daniel Jour	Chapter 28
Daniel Porteous	Chapter 22
Dariusz	Chapters 4, 10, 30 and 31
DarkDust	Chapter 28
David Refaeli	Chapter 24
deamentiaemundi	Chapter 26
depperm	Chapter 6
Devansh Tandon	Chapter 61
dhein	Chapter 46
dkrmr	Chapter 46
Dmitry Grigoryev	Chapter 63
Don't You Worry Child	Chapter 22
Donald Duck	Chapter 1
doppelheathen	Chapter 46
Dreamer	Chapter 63
drov	Chapters 19 and 61
DrPrItay	Chapter 30
Dunno	Chapter 63
dvhh	Chapters 19 and 46
dylanweber	Chapter 6
e.jahandar	Chapter 22
Ed Cottrell	Chapter 1
Elazar	Chapters 10 and 13
Eli Sadoff	Chapter 10
elslooo	Chapters 22 and 46
embedded_guy	Chapter 39
EOF	Chapter 46
erebos	Chapter 22
EsmaeelE	Chapters 1, 13, 38 and 45
eush77	Chapters 32 and 33
EvilTeach	Chapter 9
Faisal Mudhir	Chapters 16 and 22
Fantastic Mr Fox	Chapters 4, 9 and 22
fastlearner	Chapter 30
FedeWar	Chapters 6, 22 and 63
Filip Allberg	Chapter 24
Firas Moalla	Chapters 10 and 22
fluter	Chapter 21
foxtrot9	Chapters 18 and 22
Fred Barclay	Chapter 63
ganchito55	Chapter 29
ganesh kumar	Chapter 25
Gavin Higham	Chapters 22 and 63
gdc	Chapter 46
George Stocker	Chapter 25
Giorgi Moniava	Chapters 22, 28 and 63
gmug	Chapter 13
GoodDeeds	Chapters 15 and 42
gsamaras	Chapters 4 and 28
haccks	Chapters 6, 8, 22 and 23

haltode	Chapter 22
Harry Johnston	Chapter 22
Hemant Kumar	Chapter 22
hexwab	Chapter 29
hlovdal	Chapters 24, 30 and 63
hmijail	Chapter 28
honk	Chapters 28 and 46
hrs	Chapter 4
immerhart	Chapter 14
Insane	Chapter 5
iRove	Chapters 11 and 20
Ishay Peled	Chapters 6 and 63
Iskar Jarak	Chapter 1
Iwillnotexist Idonotexist	Chapter 4
Jacob H	Chapter 28
Jasmin Solanki	Chapter 10
jasoninnn	Chapters 6 and 12
javac	Chapter 30
Jean	Chapters 6, 28, 32, 37, 39 and 43
Jean Vitor	Chapter 20
Jens Gustedt	Chapters 1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15, 22, 23, 25, 27, 28, 30, 31, 33, 34, 35, 36, 37, 39, 40, 41, 43, 44, 46, 47, 48, 49, 55 and 57
Jeremy	Chapter 63
Jeremy Thien	Chapter 40
Jesferman	Chapters 41 and 45
John	Chapter 28
John Bode	Chapter 23
John Bollinger	Chapters 6, 10, 28, 35 and 47
John Burger	Chapter 24
John Hascall	Chapter 63
JonasCz	Chapter 1
Jonathan Leffler	Chapters 1, 2, 4, 6, 7, 9, 11, 13, 14, 17, 19, 20, 21, 22, 28, 30, 32, 33, 39, 42, 44, 45, 46, 49, 50, 53, 59, 61, 62 and 63
Jonathon Reinhart	Chapter 44
Jossi	Chapters 11, 21, 33, 37 and 56
Juan T	Chapter 1
juleslasne	Chapters 1 and 46
Justin	Chapter 30
jxh	Chapters 15, 33 and 37
Kamiccolo	Chapter 28
kamoroso94	Chapter 13
kdopen	Chapter 4
Ken Y	Chapter 3
Kerrek SB	Chapters 8 and 18
Klas Lindbäck	Chapter 7
Kusalananda	Chapter 1
L.V.Rao	Chapters 4, 10, 15, 16, 22, 43 and 52
LaneL	Chapter 22
lardenn	Chapter 21
Leandros	Chapters 1, 3, 4, 10, 24, 25, 28, 29, 30, 31 and 33
LiHRaM	Chapter 1
Liju Thomas	Chapters 10, 17 and 45
Lord Farquaad	Chapter 63

Luiz Berti	Chapter 46
Lundin	Chapter 40
madD7	Chapters 9, 41, 43 and 46
Madhusoodan P	Chapter 30
Malcolm McLean	Chapters 6, 15, 19, 20, 22, 24, 26, 29, 33, 46 and 49
Malick	Chapter 1
manav m	Chapter 14
mantal	Chapter 6
Mark Yisri	Chapters 1, 28 and 46
Martin	Chapter 28
Matthieu	Chapter 46
MC93	Chapters 1 and 37
mhk	Chapter 10
Michael Fitzpatrick	Chapter 22
MikeCAT	Chapters 4, 6, 16, 22, 25, 28 and 63
mirabilos	Chapters 31 and 32
mpromonet	Chapter 35
Nemanja Boric	Chapter 28
NeoR	Chapter 16
Neui	Chapters 43 and 46
Nitinkumar Ambekar	Chapters 22, 30 and 43
Nityesh Agarwal	Chapter 62
noamgot	Chapters 4 and 16
OiciTrap	Chapter 1
OznOg	Chapters 10, 12 and 16
P.P.	Chapters 1, 4, 6, 16, 17, 22, 28, 43, 46 and 57
pandaman1234	Chapters 10, 16, 49 and 60
Parham Alvani	Chapter 58
PassionInfinite	Chapter 53
Paul Hutchinson	Chapter 9
Paul V	Chapter 46
Paul92	Chapters 4 and 7
Peter	Chapters 4 and 28
PhotometricStereo	Chapters 13 and 42
polarysekt	Chapter 21
PSN	Chapter 1
Purag	Chapter 6
Qrchack	Chapter 37
R. Joiny	Chapter 37
Rakitić	Chapter 1
RamenChef	Chapter 56
Ray	Chapter 10
reshad	Chapters 15 and 20
Richard Chambers	Chapters 25 and 30
Rishikesh Raje	Chapter 9
Robert Baldyga	Chapter 30
Roland Illig	Chapters 51 and 63
rxantos	Chapter 63
Ryan Haining	Chapters 10 and 33
Ryan Hilbert	Chapter 1
Shog9	Chapter 19
Shrinivas Patgar	Chapter 26
Shubham Agrawal	Chapter 62

signal	Chapter 22
Sirsireesh Kodali	Chapter 52
skrtbhtngr	Chapter 1
slugonamission	Chapter 22
Snaipe	Chapter 20
sohnryang	Chapter 1
someoneigna	Chapters 22 and 25
Sourav Ghosh	Chapter 49
Spidey	Chapter 22
Srikar	Chapter 46
stackptr	Chapters 1, 4, 6, 10, 20, 22, 24, 27, 30, 33, 46 and 63
StardustGogeta	Chapter 21
still_learning	Chapter 6
Sun Qingyao	Chapter 34
syb0rg	Chapters 1, 6, 19, 20, 22, 37, 46 and 49
Tamarous	Chapter 63
tbodt	Chapter 46
techEmbedded	Chapter 63
the sudhakar	Chapter 46
thndrwrks	Chapter 22
tilz0R	Chapter 45
Tim Post	Chapter 33
tlhIngan	Chapter 1
Toby	Chapters 1, 2, 4, 5, 6, 7, 9, 10, 12, 13, 14, 15, 16, 20, 22, 28, 29, 32, 37, 42, 46, 49, 51, 52, 53, 56, 58, 59, 60 and 63
tofro	Chapters 16, 37 and 46
Turtle	Chapter 37
tversteeg	Chapters 20, 28 and 41
user45891	Chapter 28
user5389107	Chapters 4, 5 and 10
v7d8dpo4	Chapter 30
Vality	Chapters 13, 22, 54 and 56
vasili111	Chapter 1
Vin	Chapter 1
Vivek S	Chapter 46
Vraj Pandya	Chapters 1 and 37
vuko_zrno	Chapters 46 and 60
William Pursell	Chapter 20
Wolf	Chapters 4 and 6
Woodrow Barlow	Chapter 19
Wyzard	Chapter 46
Yotam Salmon	Chapter 19
Алексей Неудачин	Chapters 6, 22 and 37